

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

一套代码，三端生产：移动Web、iOS、Android
知识体系，无缝衔接：框架、工具、方法



移动 Web 前端高效开发实战

HTML5 CSS3 JavaScript Webpack
React Native Vue.js Node.js

iKcamp 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

“iKcamp团队”由沪江Web前端团队中热爱原创和翻译的小伙伴发起，成立于2016年7月，“iK”代表布兰登·艾克（JavaScript之父）。追随JavaScript这门语言所秉持的精神，崇尚开放和自由的我们一同工作、分享、创作，等候更多有趣、跳动的灵魂。

本书由“iKcamp团队”制作完成，主要成员如下（排名不分先后）：

陈达孚，2015年研究生毕业于香港中文大学计算机专业，现为沪江Web前端架构部成员，主要进行前端新技术调研实践工作，曾在FDCon 2017上分享“基于React Native三端融合的应用和实践”。

朱会震，十年Web开发经验，曾任CSDN架构师，负责多个核心产品的研发工作。近几年专注于移动Web开发、前端工程化、高效能等方面的研究。现就职于沪江，负责沪江网校Web前端开发和管理工作。

哈志辉，沪江CCtalk产品线前端架构师，有过多年的前后端开发及架构经验。在前后端分离、Webpack构建、React单页应用及自动化等方面有丰富的经验。

干琨，曾就职于大众点评等互联网公司，现就职于沪江学金网络。React忠实爱好者，喜欢捣腾新技术，信奉“没有最好的技术，只有最合适的技术”。

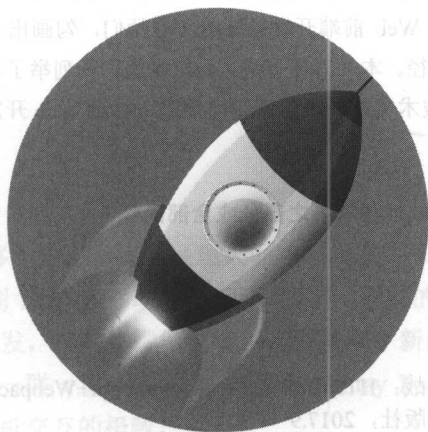
戴亮，近十年前后端开发经验。2014年加入沪江Web前端架构组，负责UI组件、移动打包平台等模块开发，推动Node.js前后端分离方案的落地。曾在GMTC 2017上分享“沪江基于Node.js大规模应用实践”。

严明坤，2003年开始从事网站开发工作，2007年赴上海发展并专注于前端领域，曾就职于盛大网络，现就职于沪江。

易未来，原沪江Web前端架构师，现就职于万达网络科技集团资深开发工程师。多年前后端开发经验，现专注于前端开发，先后在EMC、携程、沪江从事相关开发及管理工作。

周遥，《HTML 5网页开发实例详解》作者，先后在盛大网络、大众点评网就职，从事相关开发及管理工作，现为沪江Web前端横向负责人。

内容简介



移动 Web 前端高效开发实战

HTML5 CSS3 JavaScript Webpack
React Native Vue.js Node.js

iKcamp 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

移动互联网的兴起和快速普及,给前端开发人员带来了前所未有的新机遇。移动 Web 前端技术作为整个技术链条中重要的一环,却乱象丛生。本书是一本梳理移动前端和 Native 客户端技术体系的入门实战书。

本书涵盖了移动 Web 前端开发中的各个关键技术环节,共 14 章。分别从 HTML 5、CSS 3、JavaScript 的 ECMAScript 5 和 ECMAScript 6 版本、移动端常用布局方案、MV*类新时代框架、预编译技术、性能优化、开发调试、混合式应用、单元测试、工程化等方面全面地还原一线互联网公司 Web 前端技术栈。

创作本书的初衷是帮助移动 Web 前端开发领域的工程师们,勾画出一张实用并且具体的技术图,帮助读者正确且快速地掌握学习路径。本书篇幅有限,力求精简,只列举了各技术栈中核心关键部分,包括大量基于 Web 前端的优秀开源技术类库和框架介绍,是进入移动 Web 开发领域的绝佳之选。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

移动 Web 前端高效开发实战: HTML 5+CSS 3+JavaScript+Webpack+React Native+Vue.js+Node.js/
iKcamp 著. —北京:电子工业出版社,2017.9
ISBN 978-7-121-32481-9

I. ①移… II. ①i… III. ①超文本标记语言—程序设计②网页制作工具③JAVA 语言—程序设计
IV. ①TP312.8②TP393.092.2

中国版本图书馆 CIP 数据核字 (2017) 第 194788 号

责任编辑:董 英

印 刷:三河市鑫金马印装有限公司

装 订:三河市鑫金马印装有限公司

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱

邮编:100036

开 本:787×980 1/16

印张:33.75

字数:768 千字

版 次:2017 年 9 月第 1 版

印 次:2017 年 9 月第 1 次印刷

印 数:3000 册 定价:89.00 元

凡所购买电子工业出版社图书有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系,联系及邮购电话:(010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式:010-51260888-819, faq@phei.com.cn。

前 言

国内从 2012 年开始，移动端设备流量呈现大面积爆发式增长，到 2017 年已经达到 75%，预计 2018 年这一比例会达到 79%。在移动的浪潮下，前端工程师的战场面临着一次巨大的迁移，区别于传统的 PC 端 Web 开发，移动终端给前端工程师带来了新的挑战，但更多的是机遇。令人激动的是，前端工程师是一群一直信奉“Stay Hungry, Stay Foolish”的伙伴，充满激情和活力，时刻拥抱变化，追求人机交互的极致。

恍若隔世，犹记得在 Internet Explorer 6 时代，前端工程师忙碌于兼容各种奇异浏览器，受困于职业的被误解，受限于业务场景的单一，壮志未酬的身影。时至今日，前端领域技术栈日渐宽广和深厚，向后有基于 Node.js 的大规模前后端分离实践，向前有基于 React Native 或 Weex 的多端融合方案，从 PC 端到移动端有大量优秀 MV*类框架的探索和应用，身后有诸如 Webpack 或 Rollup 的工程化支持。作为这个时代身处移动变革中的前端工程师的我们，应保持信仰，不断学习前进，努力构建一个精彩的多元化世界。

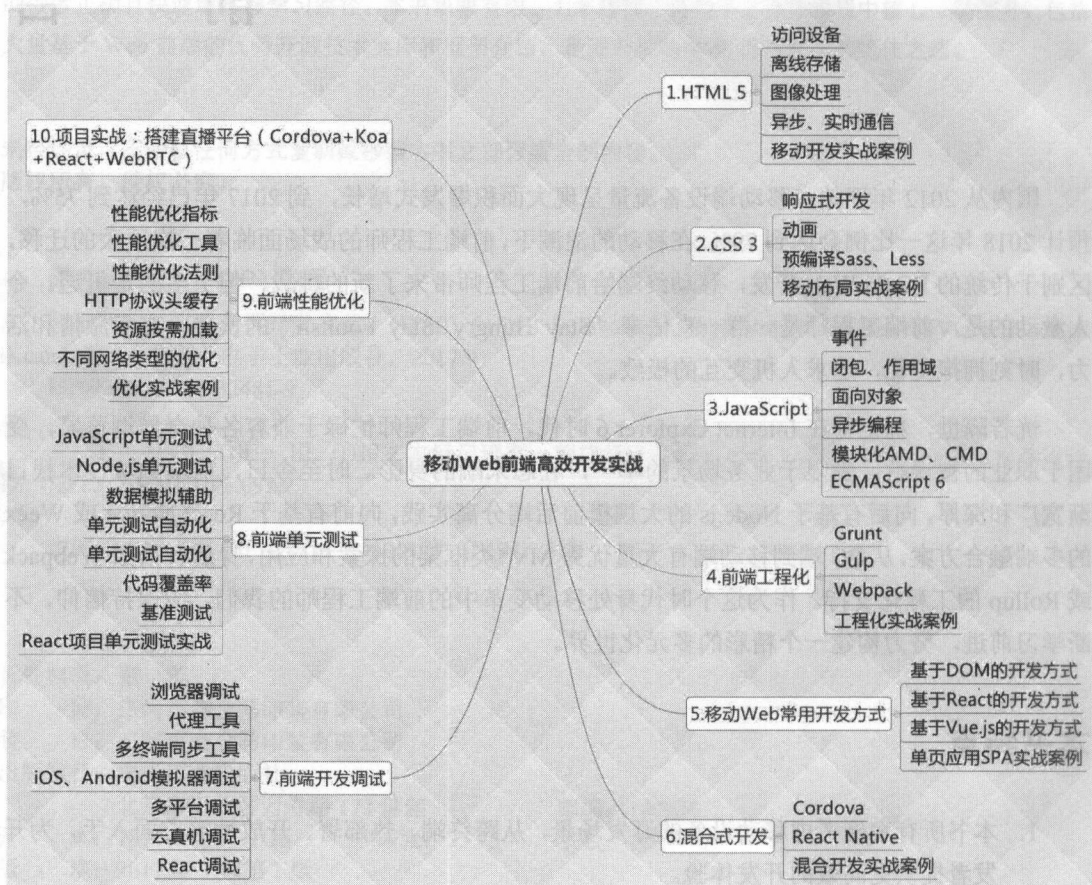
本书特点

1. 本书所有案例考虑移动设备的开发场景，从跨终端、热部署、开放性等方面入手，为开发者提供更高效的开发体验。
2. 本书内容技术新颖、与时俱进，结合时下最热门的技术，如 HTML 5、CSS 3、ES6、Node.js、React、Webpack、Cordova、React Native、Mocha，让读者在学习移动 Web 前端技术的同时，了解并熟识更多相关的世界先进开源解决方案。
3. 本书案例贴近项目开发真实环境，结合大量成熟第三方组件和框架的使用，帮助读者快

速找到问题的最优解决方案。

4. 本书的目的就是帮助读者形成思维方法论和构建牢固的知识体系，不管是移动 Web 还是 Native 客户端，还是跨终端、跨平台，都能在本书中找到合适的技术方案。

本书的技术体系



本书主要作者（排名不分先后）

本书由 iKcamp 团队制作完成，主要成员如下。

陈达孚，2015 年研究生毕业于香港中文大学计算机专业，现为沪江 Web 前端架构部成员，主要进行前端新技术调研实践工作，曾在 FDCon 2017 上分享“基于 React Native 三端融合的应用和实践”。

朱会震，10 年 Web 开发经验，曾任 CSDN 架构师，负责多个核心产品的研发工作。近几年专注于移动 Web 开发、前端工程化、高效能等方面的研究。现就职于沪江，负责沪江网校 Web 前端开发和管理工作。

哈志辉，沪江 CCtalk 产品线前端架构师，有过多年的前后端开发及架构经验。在前后端分离、Webpack 构建、React 单页应用及自动化等方面有丰富的经验。

干琚，曾就职于大众点评等互联网公司，现就职于沪江学银网络。React 忠实爱好者，喜欢研究新技术，信奉“没有最好的技术，只有最合适的技术”。

戴亮，近十年前后端开发经验。2014 年加入沪江 Web 前端架构组，负责 UI 组件、移动打包平台等模块开发，推动 Node.js 前后端分离方案的落地。曾在 GMTC 2017 上分享“沪江基于 Node.js 大规模应用实践”。

严明坤，2003 年开始从事网站开发工作，2007 年赴上海发展并专注于前端领域，曾就职于盛大网络，现就职于沪江。

易未来，原沪江 Web 前端架构师，现任职万达网络科技集团资深开发工程师。多年前后端开发经验，现专注于前端开发，先后在 EMC、携程、沪江从事相关开发及管理工作。

周遥，《HTML 5 网页开发实例详解》作者，先后在盛大网络、大众点评网就职，从事相关开发及管理工作，现为沪江 Web 前端横向负责人。

本书读者

- 移动 Web 开发初学者和前端爱好者
- APP 的 Native 客户端开发人员
- 网页和移动网页从业人员
- 从事后端开发对前端有兴趣的人员
- 大中专院校的学生
- 可作为各种培训学校的入门教程

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 [下载资源](#) 处下载。
- **提交勘误：**您对书中内容的修改意见可在 [提交勘误](#) 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 [读者评论](#) 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32481>



目 录

第 1 章 初识移动 Web 前端	1
1.1 移动 Web 前端史	1
1.1.1 Web 开发的变迁	2
1.1.2 移动 Web 与 HTML 5 不得不说的关系	3
1.1.3 移动 Web 与原生应用的优劣势	5
1.2 移动 Web 前端现状与未来	6
1.2.1 移动 Web 的现状	6
1.2.2 您需要掌握的知识体系	7
1.2.3 技术拐点与未来趋势	10
1.3 常见问题	11
1.3.1 移动 Web 前端开发有前景吗	11
1.3.2 PC Web 和移动 Web 开发区别	12
1.4 本章小结	13
第 2 章 移动 Web 开发环境搭建	14
2.1 Visual Studio Code 免费跨平台编辑器	14
2.2 使用 Node.js	16
2.2.1 Node.js 的用途	16
2.2.2 安装和调试 Node.js	17
2.2.3 什么是 NPM	19
2.2.4 Web 代理工具 NProxy	22

2.2.5 HTTP 服务器 http-server.....	24
2.3 本章小结	25
第 3 章 HTML 5 必会实际常用特性.....	26
3.1 新的语义	26
3.1.1 新元素的到来.....	26
3.1.2 表单的增强应用	28
3.1.3 使用音频和视频.....	32
3.2 访问你的设备	34
3.2.1 定位当前地理位置.....	35
3.2.2 实战演练: 调用摄像头拍个照.....	37
3.2.3 实战演练: 在手机上实现摇一摇	41
3.3 离线和存储	44
3.3.1 实战演练: 搭建一个简单的离线应用	44
3.3.2 离线之后资源该如何更新——Service Worker	47
3.3.3 LocalStorage 与 SessionStorage	48
3.3.4 实战演练: 利用 IndexedDB 实现便签管理.....	51
3.4 图像效果	56
3.4.1 使用 Canvas 绘制一个简单的饼图	56
3.4.2 使用 SVG 实现奥运五环.....	58
3.4.3 WebGL 带来了 3D 图像功能	60
3.5 不一样的通信	62
3.5.1 PostMessages	62
3.5.2 XMLHttpRequest Level 2	65
3.5.3 Server Sent Event	69
3.5.4 WebSocket	72
3.5.5 WebRTC	73
3.6 其他常用特性	77
3.6.1 History API 与单页应用.....	77
3.6.2 Drag 和 Drop 介绍.....	79
3.6.3 利用 Web Workers 加速应用计算.....	81
3.6.4 利用 Performance API 分析网站性能	82

3.7 本章小结	85
第4章 CSS 3 必会实战技巧	86
4.1 认识 CSS 3	86
4.1.1 什么是 CSS 3	87
4.1.2 移动 Web 的 CSS 3 现状	89
4.1.3 用 Modernizr 检测浏览器是否支持 CSS 3	93
4.2 选择器	96
4.2.1 常见选择器	97
4.2.2 伪类和伪元素	99
4.2.3 优先级和权重	104
4.3 响应式开发	106
4.3.1 常见设备的宽高	106
4.3.2 Flex 弹性盒布局	108
4.3.3 媒体查询 (Media Query)	112
4.3.4 用 rem 开发响应式设计	115
4.3.5 多列 (Multiple Columns)	119
4.4 动效	122
4.4.1 转换 (Transform)	122
4.4.2 过渡 (Transition)	126
4.4.3 动画 (Animation)	128
4.5 常用特性	131
4.5.1 开放字体格式 (WOFF)	131
4.5.2 背景 (Backgrounds)	134
4.5.3 颜色 (Color)	138
4.5.4 文字效果 (Text Effects)	141
4.5.5 边框 (Border)	144
4.6 预编译	147
4.6.1 Less 介绍和安装	147
4.6.2 Less 使用	150
4.6.3 Sass 介绍和安装	154
4.6.4 Sass 使用	156

4.6.5	Compass 的安装和使用	160
4.7	本章小结	163
第 5 章 JavaScript 关键语法及使用技巧		164
5.1	理解 JavaScript	164
5.1.1	语言基础	165
5.1.2	函数和参数	168
5.2	事件	171
5.2.1	事件概述	171
5.2.2	事件委托	172
5.2.3	移动端事件	175
5.3	作用域、闭包和 this	178
5.3.1	使用 let 实现块级作用域	178
5.3.2	闭包	180
5.3.3	采用 call、apply、bind 改变 this	182
5.4	面向对象	184
5.4.1	原型和原型链	184
5.4.2	Mixin 模式	186
5.4.3	ECMAScript 6 的 Class 和 Extends	188
5.5	异步编程	189
5.5.1	AJAX 中的 Callback 回调函数	189
5.5.2	Promise 模式	190
5.5.3	生成器 Generator	192
5.6	模块化	194
5.6.1	为什么需要模块化	195
5.6.2	AMD 和 CMD 规范	197
5.6.3	ECMAScript 6 标准的模块支持	205
5.7	ECMAScript 6 其他常用功能	207
5.7.1	基础数据类型的扩展	207
5.7.2	使用解构赋值来简化代码	210
5.7.3	使用 Babel 插件提前使用新特性	212
5.8	本章小结	215

第 6 章 HTML 5 移动开发实战	216
6.1 在地图上显示行走轨迹	216
6.2 仿原生相册	220
6.2.1 实现相册初始展示页	221
6.2.2 通过手势操作控制相片	222
6.3 使用 Socket.IO 制作小型聊天室	224
6.3.1 前端 HTML+JavaScript 实现聊天界面	225
6.3.2 服务器端 Node.js 监听连接	227
6.4 移动端拍照上传实践	228
6.4.1 前端 HTML+CSS+JavaScript	229
6.4.2 服务器端 Node.js	232
6.5 利用 Microdata 进行 SEO 优化	232
6.5.1 认识 Microdata	233
6.5.2 提升网页 SEO 效果	233
6.6 制作一个带字幕的视频播放器	237
6.7 使用 Pixi.js 制作“抓住开学君”游戏 (Canvas+WebGL)	242
6.8 用 Canvas 制作刮刮卡	248
6.9 实战演练: 实现 3D 全景效果	251
6.9.1 需要的 CSS 3 特性	251
6.9.2 实现原理	251
6.9.3 实现代码	253
6.10 本章小结	255
第 7 章 移动网页样式布局实战	256
7.1 静态布局的实际应用	256
7.1.1 设计活动页面静态布局	257
7.1.2 静态布局在移动端上的自适应	257
7.2 水平居中与垂直居中实战	259
7.2.1 水平居中	259
7.2.2 自适应块级元素水平居中	260
7.2.3 行内元素垂直居中	261
7.2.4 块级元素的垂直居中	263

7.2.5	基于视口单位的解决方案	264
7.2.6	基于 Flexbox 的解决方案	265
7.3	栅格系统实现响应式列表	267
7.3.1	实现栅格布局	267
7.3.2	栅格布局的原理	269
7.4	Flex 多栏布局实战	269
7.5	实战演练: 沪江网校首页 rem 布局实践	272
7.6	实战演练: 侧边栏的滑进滑出效果	276
7.7	实战演练: 模拟原生的页面切换效果	280
7.7.1	实现页面切换过渡效果	280
7.7.2	模拟切换原理解析	283
7.8	提高 Web 动画的性能实战	284
7.8.1	使用 CSS 3 动画	284
7.8.2	使用高性能的 JavaScript 动画	285
7.9	实战演练: 课程分类列表实战	286
7.9.1	实现主页结构	287
7.9.2	响应式 CSS 实现 (Compass)	289
7.9.3	添加页面动态效果	293
7.10	本章小结	294
第 8 章	前端工程化实战	295
8.1	前端工程化	295
8.1.1	前端工程化的必要性	296
8.1.2	前端工程化的发展史	298
8.2	工程化入门 Grunt	300
8.2.1	安装和配置	301
8.2.2	Grunt 插件	302
8.2.3	实战演练: 使用 Grunt 开发一个简易相册	305
8.3	使用 Gulp 构建一个 ECMAScript 6 和 Sass 应用	309
8.3.1	安装和配置	309
8.3.2	预处理任务	310
8.3.3	实战演练: 采用 ECMAScript 6 开发一个 Markdown 编辑器	312

8.3.4	代码检查任务	315
8.3.5	代码合并、压缩、重命名任务	317
8.3.6	监听文件变化自动构建	318
8.4	实战演练：使用 Webpack 构建一个 React 应用	320
8.4.1	安装和配置	320
8.4.2	常用的加载器和插件	323
8.4.3	缓存控制	327
8.4.4	简化模块引用	330
8.4.5	异步模块加载	332
8.4.6	使用 Source Map 调试代码	335
8.5	本章小结	338
第 9 章	移动 Web 常用开发方式实战	339
9.1	基于 DOM 的开发方式	339
9.1.1	使用 Zepto 和前端模板开发简单备忘录	340
9.1.2	解决原生单击事件的缺陷	341
9.1.3	为何抛弃掉 Zepto	343
9.2	基于 React 的开发方式	345
9.2.1	使用 JSX 语法创建 React 组件	345
9.2.2	在实践中掌握 React 生命周期	348
9.2.3	实现组件间通信	353
9.2.4	实现组件关注分离	355
9.2.5	实战演练：运用组件化方式开发一个备忘录	358
9.2.6	如何管理应用的状态	364
9.2.7	添加动画效果	366
9.2.8	提高 React 组件性能	369
9.3	基于 Vue.js 的开发方式	372
9.3.1	实战演练：开发一个简单的备忘录应用（Vue.js 2.0）	372
9.3.2	管理应用的状态及实现组件间的通信	375
9.3.3	添加动画效果	379
9.4	打造单页应用 SPA	381
9.4.1	单页应用的优势是什么	382

9.4.2	实战演练：实现一个单页路由	382
9.4.3	实战演练：使用 React 开发一个简单的单页应用	384
9.4.4	单页应用的状态管理	386
9.5	本章小结	388
第 10 章	混合式开发实战	389
10.1	为什么需要混合式开发	389
10.1.1	混合式开发种类	389
10.1.2	混合式开发的优势	390
10.1.3	选择合适的混合式开发方案	391
10.2	Cordova 开发入门	392
10.2.1	JavaScript 和 Native 互相调用	392
10.2.2	Cordova 介绍和安装	394
10.2.3	Cordova 开发使用	397
10.3	React Native 实战	400
10.3.1	React Native 简介	400
10.3.2	React Native 样式和布局	402
10.3.3	React Native 组件概念	404
10.3.4	简单组件实战	404
10.3.5	复合组件实战	405
10.3.6	第三方组件实战	406
10.3.7	常用 API 实践	407
10.4	实战演练：用 React Native 开发新闻阅读应用	410
10.4.1	React Native 的工程项目结构一览	410
10.4.2	列表页	411
10.4.3	新闻评论页	414
10.4.4	新闻展示页	414
10.5	本章小结	415
第 11 章	前端开发调试实战	417
11.1	浏览器调试	417
11.1.1	Chrome 开发者工具	418

11.1.2	Safari 开发者工具	421
11.2	代理工具	424
11.2.1	Mac OS 下 Charles 的用法	424
11.2.2	Windows 下 Fiddler 的用法	426
11.3	多终端同步工具	428
11.3.1	多设备浏览器同步测试工具 BrowserSync	429
11.3.2	双向自动刷新样式工具 Emmet LiveStyle	431
11.4	模拟器调试	432
11.4.1	Android 模拟器调试	432
11.4.2	iOS 模拟器调试	434
11.4.3	在线模拟器 Manymo	436
11.5	多平台调试	437
11.5.1	网站响应式设计测试工具 Ghostlab	437
11.5.2	移动端 Web 开发调试工具 Weinre	439
11.5.3	JavaScript 远程调试和测试工具 Vorlon.JS	442
11.6	云真机调试	444
11.6.1	浏览器兼容性云端测试应用 BrowserStack	444
11.6.2	Web 端移动设备管理控制工具 STF	446
11.6.3	多浏览器兼容性测试平台 F2etest	448
11.7	React 调试	452
11.7.1	React Developer Tools	452
11.7.2	Redux DevTools	455
11.8	本章小结	458
第 12 章	前端单元测试实战	459
12.1	JavaScript 单元测试框架 Jasmine 实战	459
12.2	使用 Mocha 和 Chai 在 Node.js 进行单元测试	462
12.3	使用 Sinon 辅助单元测试	465
12.4	使用 Karma 自动化单元测试	468
12.5	使用 Istanbul 计算代码覆盖率	470
12.6	使用 Benchmark.js 进行基准测试	473
12.7	实战演练: React 版备忘录项目的完整单元测试	475

14.1.2	启动项目	501
14.1.3	Cordova 打包	502
14.2	直播平台功能预览	502
14.2.1	直播流程	503
14.2.2	直播核心页面	503
14.3	页面的布局 and 结构	504
14.3.1	首页	504
14.3.2	发起直播页面	505
14.3.3	观看直播页面	505
14.4	搭建 WebRTC 服务端——Koa	506
14.5	实现多人在线直播功能	512
14.6	实现弹幕客户端与服务端通信	517
14.6.1	客户端与服务端通信的过程	517
14.6.2	客户端代码设计——React	518
14.6.3	服务端代码设计	520
14.7	本章小结	521

1

第 1 章

初识移动 Web 前端

移动 Web 开发属于前端开发中的一个子集，专指移动设备上的 Web 前端开发工作。顺应着互联网和硬件的升级，以 iPhone 出现后的移动设备爆发为起点，用户每天的上网方式发生了很大改变。原本固定地点的 PC（Personal Computer，个人计算机）互联变为了如今随时随地的移动互联，几乎人人都拥有一台属于自己的智能手机，时时刻刻与世界的任何一个角落发生着联系。

传统前端开发者的日常工作内容也悄悄地发生了变化，开始由 PC 战场迁移至移动战场，同时，开发使用的技术也同步进行着一轮又一轮的升级换代，Web 前端领域尤其明显。本章将从移动 Web 的发展历史、现在和未来三个角度帮助读者对其形成一个基本的认识。

1.1 移动 Web 前端史

移动 Web 前端虽然兴起时间不长，发展却非常迅速，作为一名 Web 前端开发者，了解移动 Web 前端的发展是很有必要的。

1.1.1 Web 开发的变迁

Web 的应用架构最早是在 1989 年由英国人 Tim Berners-Lee 提出的,在提出的第二年,第一台 Web 服务器诞生。此时的 Web 浏览器,只能展示静态的 HTML 内容,包含简单的文本和图像。

1995 年,Java 语言问世,Web 端静态内容大批量地向动态内容迁移,这个变革是巨大的。随后,Microsoft 发布了 Internet Explorer 4.0,其使用的 DHTML 技术,可以使浏览器获得更好的展示效果。

下面通过一组苹果官网不同时间节点的截图来对比 Web 早期的发展变化,分别为 1992 年、1997 年、2000 年和 2017 年,如图 1.1、图 1.2、图 1.3 和图 1.4 所示。

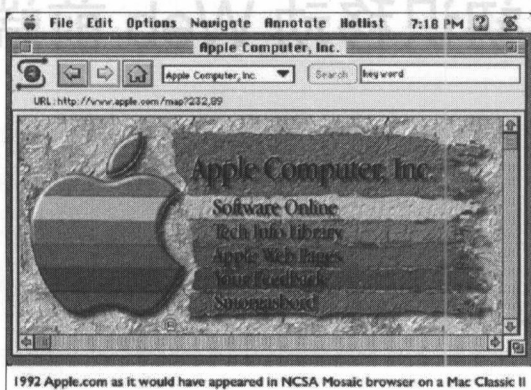


图 1.1 1992 年的苹果网站



图 1.2 1997 年的苹果网站



图 1.3 2000 年的苹果网站



图 1.4 2017 年的苹果网站

Web 开发基本上可以分为三大阶段:

- 静态技术阶段
- 动态技术阶段
- 后 Web 2.0 阶段

前面已经简单介绍了静态和动态两个阶段，Web 2.0 概念来自于 2004 年的一场头脑风暴，其代表着 Web 未来的发展趋势更注重用户之间交流互动，用户既是网站内容浏览者，也是网站内容的制造者。用户从单纯的“只读”模式向“写入”模式过渡，Web 得到了共同建设的快速发展，内容量迅速扩充。同时，Web 应用的功能也开始从单纯的站点向复杂应用转变，前端开发者的工作也不再局限于文字和图片的排列布局。加之 HTML、JavaScript 和 CSS 不断的版本升级迭代，一刻不停地适应着终端种类的纷繁复杂，作为仅存不多的“Write Once, Run Anywhere”技术栈，一个重前端的时代开始真正从沉睡中苏醒。

1.1.2 移动 Web 与 HTML 5 不得不说的关系

HTML 5 这个词汇对于 Web 前端开发者来说，应该是再熟悉不过了。这其中的数字代号“5”表示超文本标记语言 HTML 的版本号，也就是第 5 次重大修改。虽然 HTML 5 技术早在 2011 年就已经被各大浏览器厂商所支持，但是该标准真正制定完成发布时间却是在 2014 年 10 月 29 日，这表示一个新的 Web 时代已经到来。

HTML 5 带着很多使命来到这个世界，具有以下这些特性。

- **语义化**：拥有更加丰富的标签，对微数据、微结构等有着非常友好的支持，赋予网页更好的意义和结构。
- **本地存储**：基于该技术开发的网页应用拥有更短的启动时间，更快的联网速度，甚至可以做到离线使用。
- **设备兼容**：HTML 5 为开发者提供了非常丰富的 API，让开发者能够在功能上有更好的体验和优化选择。
- **连接特性**：Server-Sent Event 和 WebSocket 技术，使得连接工作效率更加有效，特别是在实时聊天和网页游戏方面，大大提高了用户体验。
- **多媒体**：网页标签自身支持音频和视频的播放，完全打破了依赖 Flash 的困局，为开发降低成本提高效率的同时，用户体验也得以提升。
- **图形特效**：HTML 5 提供了诸如 Canvas、WebGL 等图形和三维功能，使普通网页也能呈现出惊人的视觉效果。

除了以上这些让人眼前一亮的自身特性之外，HTML 5 能够在当今 Web 开发上大行其道，还

有一个很重要的原因是移动互联网的兴起。移动设备鉴于自身固有特点,在 Web 开发上跟 PC 端有着很大的差别,而相当一部分 HTML 5 特性,在移动设备上能有更大的施展空间。可以说,移动 Web 开发已经离不开 HTML 5,同样,Web 技术的迅猛发展也无法离开移动设备的普及。

下面简单列举一些移动 Web 开发常用的 HTML 5 技术。

1. 视口控制 (Viewport)

当设计师在设计网页时,一般都会按照一个固定的宽度设计,比如在 PC 端是 1000 像素或者 1200 像素等,在移动端是 640 像素或者 750 像素等。然而当这些网页在移动设备上浏览时会显示不完整,设备的宽度远远不够。拿 iPhone 6 来举例,其视口宽度是 375 像素,完全满足不了网页需求。

为了弥补这一点,移动设备上浏览器会把视口放大,一般是 980 像素或 1024 像素。但这样带来的后果是浏览器会出现横向滚动条,因为设备实际可视区域比浏览器自设的这个宽度要小很多。为了解决这个问题,就需要引入 Viewport 属性。Viewport 属性通过一个 Meta 标签引入,实例代码如下:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0, user-scalable=0">
```

上述代码的作用是让当前 Viewport 的宽度等于设备的宽度,同时不允许用户手动缩放。

2. 媒体查询 (@media)

CSS 3 引入了一个新的操作表达式,被称为媒体查询。其允许开发者基于设备的不同特性来应用不同的样式申明。其中,通过对视口宽度的判断,对网页输出不同的展示效果。比如在 iPhone 7 下默认字号为 12 像素,而在 iPhone 7 Plus 下默认字号采用 14 像素。如设备是 iPad,可选择采用多列布局展示等。

3. 音视频播放 (Audio/Video)

在 HTML 5 之前,要想在网页上播放一段音频或视频,需要借助于浏览器插件,插件通常采用 Flash 技术实现。这在 PC 设备上还勉强可以接受,但在移动设备上如果也动辄就需要安装插件,用户体验有多差可想而知。而借助于 HTML 5 的音频和视频标签,就可以轻松实现播放功能。

除此之外,还有各种布局方式,以及丰富的设备 API,这些在后续的章节中都会向读者一一介绍。

1.1.3 移动 Web 与原生应用的优劣势

移动 Web 指的是以移动端浏览器为载体面向网页的开发, 这种应用或服务一般需要通过一个 URL 来打开。而原生应用, 也就是大家所熟悉的多数 APP 的开发模式, 其针对不同的操作系统采用不同的开发语言和框架, 专门针对某一类设备而研发。这两种开发方式并没有孰好孰坏之分, 各有优劣, 本节将对这些优势和劣势逐一剖析。

1. 移动 Web 的优势

- 跨平台: 网页应用运行在浏览器上, 不直接跟系统打交道, 只要系统安装了浏览器, 就可以打开该应用。
- 开发成本低: 因为没有平台问题, 开发者不需要掌握多种开发语言和框架, 只需要一个开发团队, 就可以完成所有移动设备的前端开发工作。
- 更容易迭代: 应用所有资源都在服务端, 不需要用户主动安装更新就可以实现产品的升级迭代。

2. 移动 Web 的劣势

- 功能有限: 因为没有直接跟系统对接, 只能使用浏览器提供的部分功能, 很多硬件设备独特功能无法使用。
- 操作体验欠佳: 由于运行在浏览器之上, 用户的操作并非由系统直接接收并响应, 再加上浏览器质量参差不齐, 操作体验势必有所下降。
- 无法离线使用: 虽然 HTML 5 提供了离线存储功能, 但并不代表用户在首次访问应用时, 本地已经存在。
- 很难被发现: 用户获取 APP 的方式一般通过前往应用商店下载, Web APP 并不具备在商店展示的条件。

3. 原生应用的优势

- 功能完善: 几乎具有设备所有功能的访问权限, 可以满足用户的各种需求。
- 体验更好: 速度快、性能高, 使得原生应用的用户体验更具优势。
- 可离线使用: 由于原生应用的所有程序代码和静态资源在用户安装时已经下载到本地, 即便在断网的情况下, 用户也可以进行部分操作。
- 发现机会高: 无论是第一次下载(从应用商店), 还是再一次使用(从设备图标打开), 原生应用的机会都远大于 Web 应用。

4. 原生应用的劣势

- 开发成本高: 有多少操作系统, 就需要开发多少套应用程序, 不仅开发成本很高, 维护成

本也不容忽视。

- 迭代不可控：首先更新上线需要应用商店的审核，其次用户何时升级也是完全不受控制的。
- 内容限制：各应用商店都有自己的规范条例，原生应用的功能和内容需要完全符合这些条例才允许上架。

鉴于 Web 应用和原生应用各自的优劣势，已经有越来越多的 APP 开始走向混合开发的模式，即原生和 Web 同时存在。原生部分为用户提供更好的使用体验，Web 部分可以实现更为快速的迭代更新。

1.2 移动 Web 前端现状与未来

我们这一代人是非常幸运的，处在一个互联网快速发展的时代，也是一个信息和科技发生重大变革的时代。前端作为这些信息的入口，变得越来越重要。

1.2.1 移动 Web 的现状

在 21 世纪，互联网日新月异，移动互联网蓬勃发展，智能终端设备已经走进千家万户。甚至于，要想“幸福”地生活下去，或多或少都会接触到智能设备，都需要了解和使用 APP 功能。Web 应用作为 APP 的一种存在形式，也必将受到越来越高的重视。

对于移动端开发，先来看一张示意图，如图 1.5 所示。

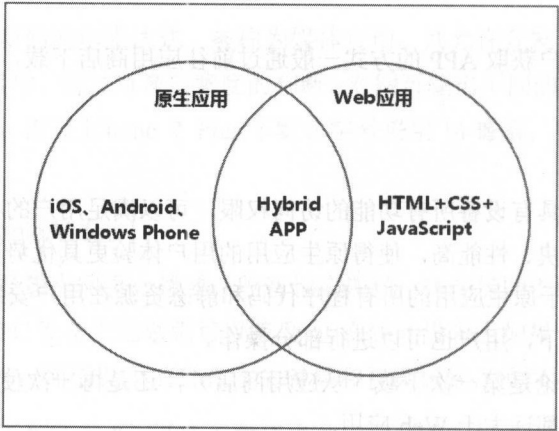


图 1.5 移动开发示意图

在前端开发这个大领域，移动 Web 前端开发虽然被认可的时间比较晚，但是发展势头比较迅

猛。在 HTML 5 的带动下，出现了一系列新的标准和技术，前端开发框架也如同雨后春笋般涌现了出来。诸如 Google、Facebook、阿里巴巴、腾讯等这些互联网巨头率先嗅到移动 Web 前端开发的重要性，开始大规模地对 Web 前端进行重构，也有越来越多的公司加入混合应用开发的队伍中来。

虽然移动 Web 前景如此明朗，但仍有很多问题需要解决。

1. 浏览器种类繁多、参差不齐

除了系统原生浏览器，还有很多第三方浏览器，如 UC、百度、腾讯、360、遨游等。这些浏览器对 HTML 5 的支持程度不一，对网页的渲染和交互也各有不同，除此之外还有一些浏览器性能堪忧，这大大增加了 Web 前端开发的成本。

2. 网速仍然是性能瓶颈

在 PC 时代，网速是困扰用户和开发者的一大难题，到了移动时代，这个问题更是被加倍放大。移动设备所处的网络环境差，是客观存在的普遍现象，无论是 2G、3G 还是 4G，网速都是阻碍 Web 应用发展的一个瓶颈。很多时候，开发者都要为网页快速加载做质量甚至功能上的让步。

3. 多框架造成高门槛

在众多开发技术中，Web 前端可以算是比较易学的一种。其没有服务端错综复杂的业务逻辑，也不用配置臃肿的开发环境，JavaScript 语言更是简单轻量。但是现如今，为了解决 Web 前端开发工程化、模块化、开发和维护成本等一系列问题，出现了一大批前端开发框架，别说新入门的开发者，即便是有多年经验的前端老手，想要全部掌握这些框架，也是一个艰难的任务。

无论怎样，这是一个移动 Web 前端的大好时代，有机会，也有挑战。其中暴露出的 Web 问题，随着时间的推移，终将被一一解决。

1.2.2 您需要掌握的知识体系

通过上一节了解了移动 Web 及其现状之后，本节将介绍作为移动 Web 前端工程师需要掌握的知识体系，整体结构如图 1.6 所示。

作为移动 Web 前端工程师首先必定是一名合格的软件工程师，然后才会因为精通前端领域而被认为是一名优秀的移动 Web 前端工程师。接下来将介绍图 1.6 中包含的每个知识模块的详情。

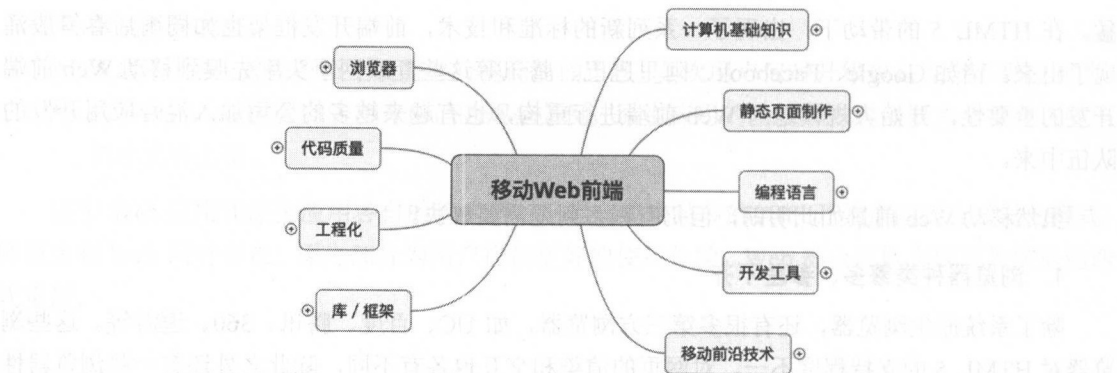


图 1.6 移动 Web 前端知识体系图

(1) “计算机基础知识”模块包含的知识体系如图 1.7 所示。

(2) “静态页面制作”模块包含的知识体系如图 1.8 所示。

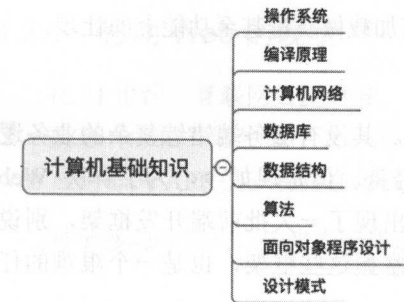


图 1.7 计算机基础知识

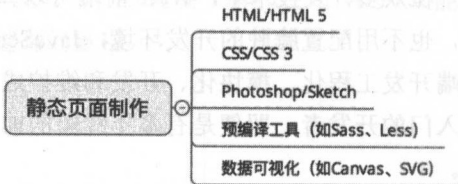


图 1.8 静态页面制作

(3) “编程语言”模块包含的知识体系如图 1.9 所示。

(4) “开发工具”模块包含的知识体系如图 1.10 所示。

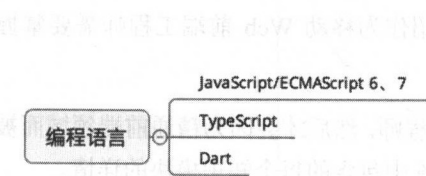


图 1.9 编程语言

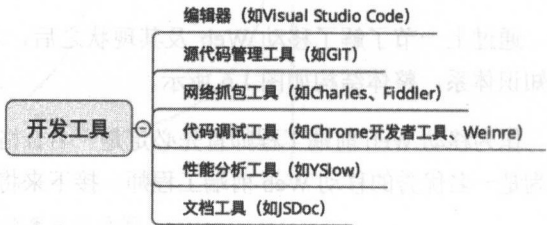


图 1.10 开发工具

(5) “浏览器”模块包含的知识体系如图 1.11 所示。

(6) “代码质量”模块包含的知识体系如图 1.12 所示。

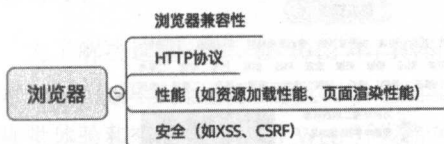


图 1.11 浏览器

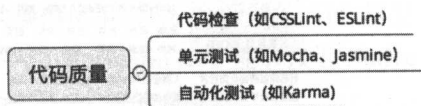


图 1.12 代码质量

(7) “工程化”模块包含的知识体系如图 1.13 所示。

(8) “库/框架”模块包含的知识体系如图 1.14 所示。

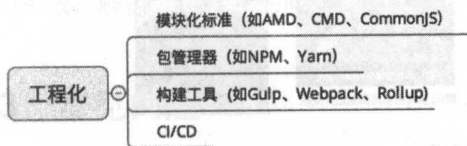


图 1.13 工程化

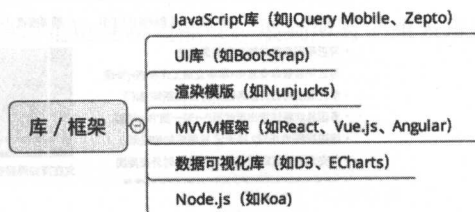


图 1.14 库/框架

(9) “移动前沿技术”模块包含的知识体系如图 1.15 所示。

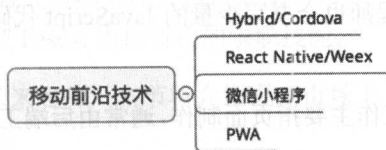


图 1.15 移动前沿技术

正是由于对 Web 前端这个职能的定义范围越来越广，能力要求越来越高，才会出现独立的 Web 前端工程师角色。Web 前端工程师虽然有时被误解为 UI 设计师，但其实静态页面制作只是其日常工作中的一小部分，其主要的工作是进行浏览器端编码。随着 Node.js 的流行，一部分后端编码逻辑也变成了 Web 前端工程师的工作内容之一。可见，计算机基础知识是一名软件工程师必须掌握的，Web 前端工程师也不例外。

1.2.3 技术拐点与未来趋势

在 2005 年以前，主流的网页如图 1.16 所示。



图 1.16 搜狐官网

注意：这是现在搜狐官网的截图，代表了 2005 年以前主流网页形态。

这类网页的特点是页面风格比较简陋，而且没有过多交互。通常，后端工程师会使用后端模板完成页面渲染。同时，后端工程师也会书写少量的 JavaScript 代码完成一些基础的页面交互，如验证表单输入信息。

这个时期的特点是，前端的工作主要指页面制作，通常由后端工程师使用后端 Web 框架完成，或者由 UI 设计师完成。

拐点来自于 Google 在 2005 年推出的 Web 邮箱 Gmail。为了提高用户体验，Google 使用了大量的 AJAX 技术，将 Gmail 实现为一个单页应用，JavaScript 作为第一编程语言在项目里被大量使用。

这段时间还是纯粹的 PC 时代，在这期间，也有 Web 前端工程师的称呼，但概念其实比较混乱。有可能指的是 UI 设计师，也有可能指的是偏 Web 开发的后端工程师。随着 PC Web 应用的复杂化，大量具备软件工程知识的开发者慢慢地转向前端领域，促进着该行业的快速发展。

随着智能手机的普及，PC 业务慢慢向移动端转移，移动端 APP 开始大面积兴起，业务版本

的快速迭代,也使得原生移动的开发方式的缺点暴露得越来越明显。除了开发成本过高,同一个 APP 需要在 iOS 和 Android 端实现两次外,最致命的缺点是每次更新都需要发版,用户需要重新安装 APP。

为了解决这些问题,在 2012 年,Hybrid 技术开始被大规模使用。Hybrid 开发的 APP 基于 Web 技术,一套代码多处运行,而且可以达到及时更新的效果。为了让 Hybrid APP 能够接近 Native APP 的视觉体验和交互体验,对 Web 开发者的能力要求也达到了一个新的高度。

Web 前端技术圈是一个开放的、充满激情的、不断扩展自身能力边界的圈子。在纵向上,Node.js 把边界扩展到服务器端开发。在横向上,React Native 试图使用 Web 技术开发 Native APP。在新兴领域,如 VR、AR、物联网等领域,也都在试图进入、制定相关标准。其他方面,如微信小程序,也使用 Web 技术开发。

看完 Web 前端的前世今生,可以发现,Web 前端的定位是希望能够扩展到所有的跟表现层相关的领域,解决人机交互的问题。

1.3 常见问题

1.3.1 移动 Web 前端开发有前景吗

当投入精力去学习一门技能的时候,尤其是要把这门技能作为自己的求职技能时,往往都需要关注技能在现在以及将来的就业市场上能否保持强劲的竞争力。这是无可厚非的,毕竟很少有人今天会去学习 Pascal,然后把 Pascal 当作自己的求职技能。

上一节讲述了 Web 前端未来趋势,本节将介绍现在市场上 Web 前端工程师的就业行情,以及 Web 前端工程师的发展方向。

2016 年某专业招聘网站技术岗位就业情况,如图 1.17 所示。2017 年春季技术岗位就业情况如图 1.18 所示。可以看到,近两年,Web 前端工程师的需求仍旧非常旺盛。同时值得庆幸的是,Web 前端行业一直在扩展自己的领域边界。前端已经不再是一个只有 HTML、CSS 和 JavaScript 的领域,读者应该保持对自身技术边界的开拓。

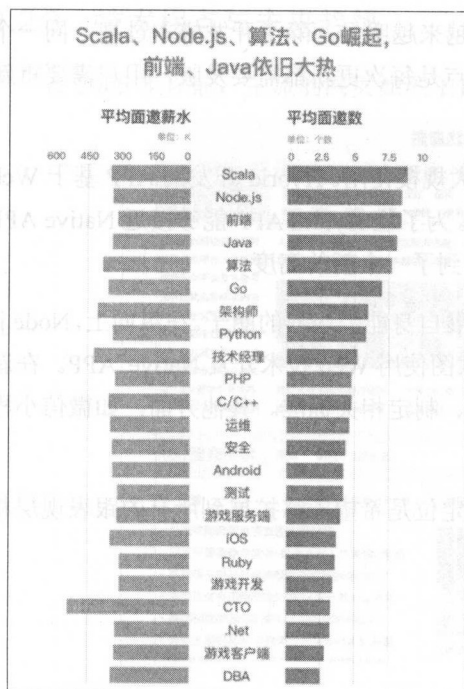


图 1.17 2016 年互联网行业技术职位就业情况

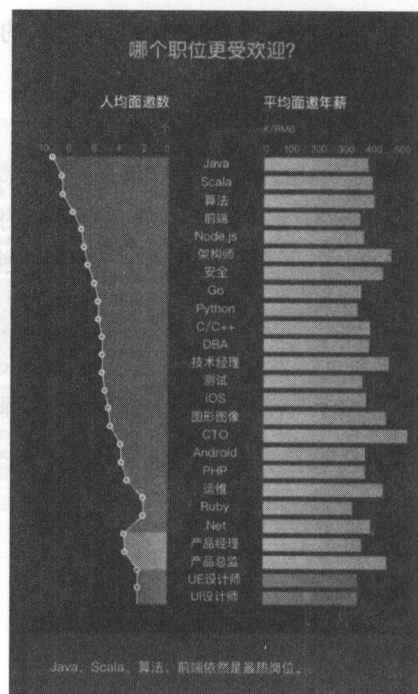


图 1.18 2017 年春季互联网行业技术职位就业需求

1.3.2 PC Web 和移动 Web 开发区别

PC Web 和移动 Web 开发所需要掌握的基本知识体系并没有太大区别，只是不同的终端（如电脑、手机）具有不同的特性。本书主要介绍移动 Web 前端开发技能，所以接下来结合本书的部分大纲来讲解需要重点关注的知识点。

- 第 3 章 HTML 5 快速入门，讲述 HTML 5 新增功能的应用，如实现手机摇一摇。
- 第 4 章 CSS 3 基础，讲述 CSS 3 新增功能的应用，如 CSS 3 动画。
- 第 6 章 HTML 5 开发实战，讲述移动端潮流技术的应用，如 3D 全景效果。
- 第 7 章移动网页样式布局实战，讲述移动端布局，如 Flex 布局、rem 布局。
- 第 10 章混合式开发实战，讲述 Hybrid 开发方式，如使用 Cordova，还介绍了 React Native 技术。
- 第 11 章实战中的开发调试，讲述移动端调试技巧，如使用真机调试。
- 第 13 章性能优化实战，讲述移动端性能优化技巧，如针对不同网络进行优化。

总而言之，如果以前从事过 PC Web 开发工作，现在转向移动 Web 开发，绝大部分知识仍旧

需要，并且依然有效。当然，还需要学习一些跟移动 Web 开发相关的新知识。

1.4 本章小结

本章从前端的发展史出发，通过与传统 Web 开发和原生开发的比较，向读者介绍移动 Web 开发独有的特性。同时还提供了一套全面的知识体系图，帮助大家形成一个初步的认识，在接下来的章节中读者可以了解到知识体系中关键的技术点和当下通用的解决方案。

2

第 2 章

移动 Web 开发环境搭建

在进行移动 Web 开发之前，搭建一个合适并且高效的环境非常关键。本章将介绍本书实例中使用的开发工具和环境，并且推荐读者使用同样的环境进行编程。本章的开发环境基于 Mac OS 系统（相关的工具同样提供了 Windows 版本）。

2.1 Visual Studio Code 免费跨平台编辑器

这里所说的 Visual Studio Code 并非微软提供的大型开发工具包 Visual Studio，而是微软在 2015 年 4 月份发布的一款能够运行在 Windows、Mac OS 和 Linux 之上的免费跨平台编辑器。优秀的性能，完备的特性，加之针对于 Web 端开发的优化和方便的调试，使其被评价为最好用的 IDE。

Visual Studio Code 官网地址为 <https://code.visualstudio.com/>，如图 2.1 所示。

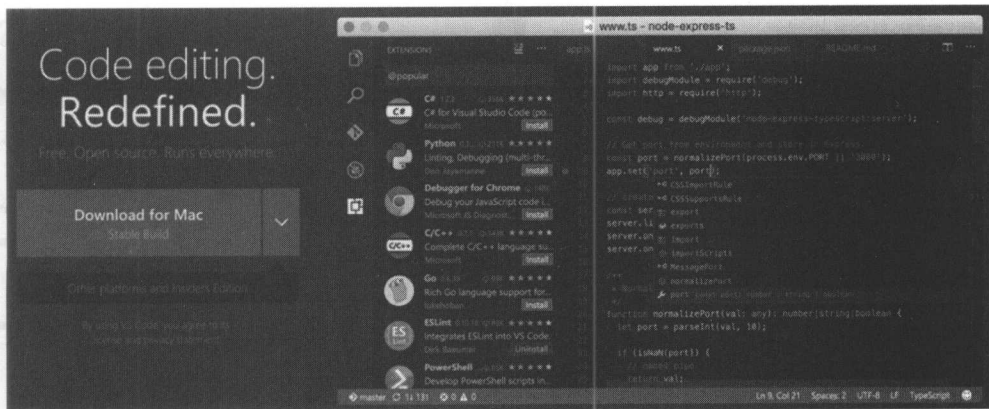


图 2.1 Visual Studio Code 官网

下载安装的步骤这里忽略，直接打开主界面，如图 2.2 所示。

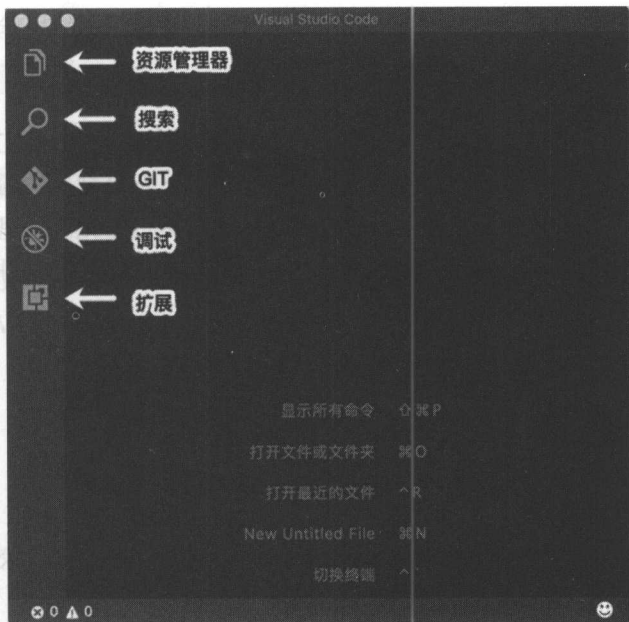


图 2.2 Visual Studio Code 主界面

同时 Visual Studio Code 提供了大量扩展插件，可以单击“扩展”图标按钮安装，也可以访问网站 <https://marketplace.visualstudio.com/VSCode> 获取，如图 2.3 所示。



图 2.3 Visual Studio Code 扩展插件

Visual Studio Code 还提供了非常方便的 Node.js 调试环境，笔者将在稍后的 Node.js 章节介绍。

2.2 使用 Node.js

Node.js 在 2009 年由 Ryan Dahl 推出，早期的 Node.js 更像是一种前端工程师的“玩具”，因为 JavaScript 也能开发后端应用。历经 8 年多的发展，Node.js 已经被用在国内外各大互联网公司的线上重要应用中，像国外的 Google、Uber、PayPal 等，国内的腾讯、阿里巴巴、百度等。还在 NASA（美国国家航空航天局）的很多项目中得到应用，包括宇航服的部分控制系统。Node.js 的重要性不言而喻。

2.2.1 Node.js 的用途

在讲 Node.js 的用途之前，先介绍 Node.js 是什么。按照官网的解释：“Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。Node.js 使用高效、轻量级的事件驱动、非阻塞 I/O 模型。”

介绍中提到的 V8 是 Google 开源的、高性能的 JavaScript 引擎，因其高效的 JavaScript 编译和执行性能，使得依赖于 V8 的 Node.js 变得更加可靠。

Node.js 用途就像前面提到的，已经渗透到各大互联网公司的重要线上应用中，同时很多创业或者中小型公司的后端甚至全部采用 Node.js 开发。当使用互联网上一项常用的服务时，说不定后面就有 Node.js 的影子。所以，Node.js 几乎能够实现一切应用，只是需要根据业务和项目选择使用。

2.2.2 安装和调试 Node.js

Node.js 官网地址为 <https://nodejs.org>，截至当前（撰写时间）Node.js 的稳定版本是 6.10.3，最新版本是 8.0.0，如图 2.4 所示。

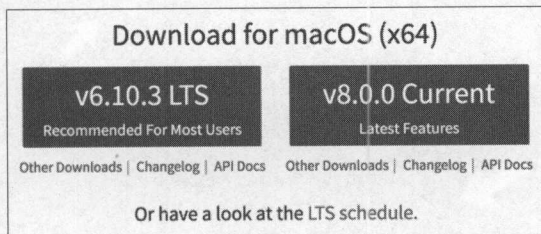


图 2.4 Node.js 官网

(1) 下载最新版本并安装，安装界面如图 2.5 所示。

(2) 打开终端验证安装是否成功，命令如下：

```
node -v
```

(3) 安装正常，则显示当前 Node.js 版本信息，如下：

```
v6.10.3
```

提示：大多数开源软件都会遇到版本问题，Node.js 也不例外。很多时候需要在本地安装多个版本的 Node.js 进行开发，这里推荐一个非常实用的 Node.js 版本管理工具 NVM (Node Version Manager)，GitHub 地址为 <https://github.com/creationix/nvm>。

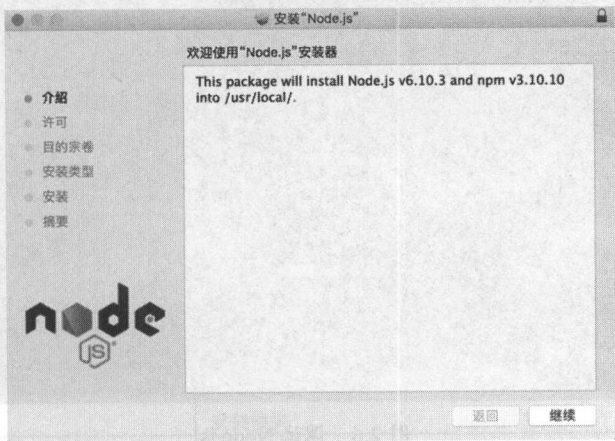


图 2.5 安装 Node.js

(4) 接着使用 2.1 节推荐的编辑器 Visual Studio Code 对 Node.js 进行调试。使用 Node.js 官方实例创建 JavaScript 文件, 代码如下:

```
01 const http = require('http');
02 const hostname = '127.0.0.1';
03 const port = 3000;
04 const server = http.createServer((req, res) => { // 创建一个 HTTP 服务
05     res.statusCode = 200;
06     res.setHeader('Content-Type', 'text/plain');
07     res.end('Hello World\n');
08 });
09 server.listen(port, hostname, () => { // 监听指定端口并启动服务
10     console.log(`Server running at http://${hostname}:${port}/`);
11 });
```

(5) 单击编辑器左侧工具栏中的“调试”按钮, 效果如图 2.6 所示。

(6) 在第 5 行代码处添加断点, 并单击左上角文字“调试”右侧的三角按钮, 开始启动调试功能, 如图 2.7 所示。

(7) 服务启动成功, 在“调试控制台”输出 HTTP 服务运行地址和端口号, 打开浏览器访问地址 <http://127.0.0.1:3000>, 将自动切换进入编辑器调试控制台。一个简单的在 Visual Studio Code 中调试 Node.js 实例介绍完毕, 读者也可以访问 <https://code.visualstudio.com/docs/nodejs/nodejs-debugging>, 查看更多高级调试功能。

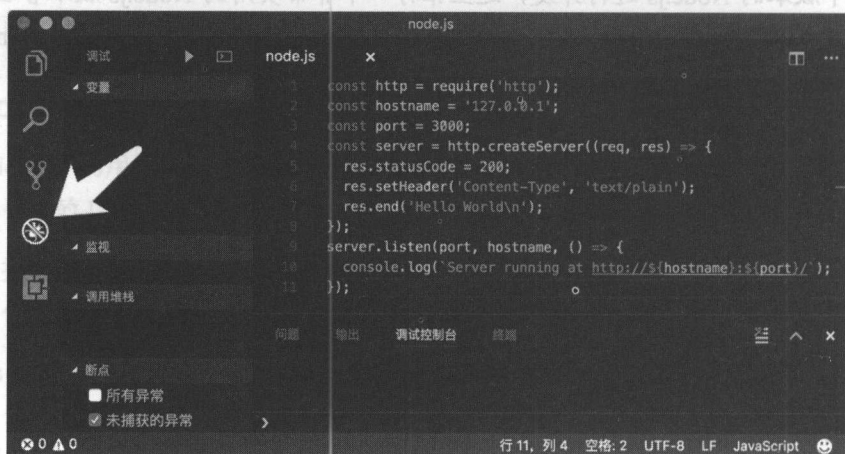


图 2.6 调试 Node.js

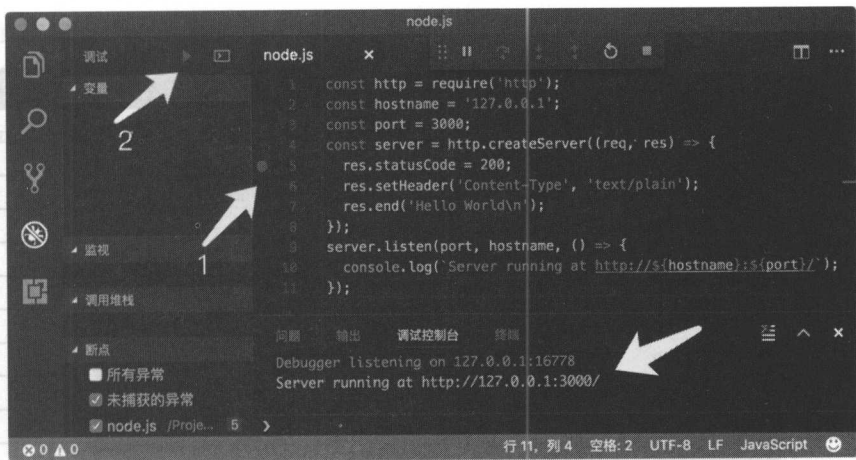


图 2.7 启动 Node.js 调试

2.2.3 什么是 NPM

NPM 全称 Node Package Manager，即 Node.js 模块管理工具，随同 Node.js 一起安装，比如安装上节介绍的 Node.js v6.10.3 时，随同的 NPM 版本是 v3.10.10。NPM 同样也有自己的官网，可以搜索社区贡献的丰富的工具包，地址为 <https://www.npmjs.com>，如图 2.8 所示。

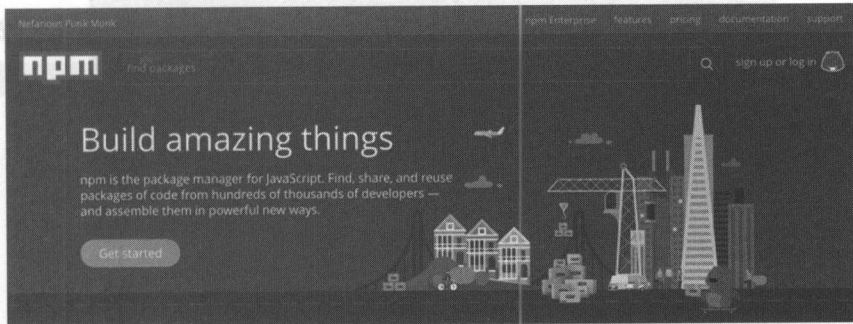


图 2.8 NPM 官网

想要顺利地使用 Node.js，学会常用的 NPM 命令必不可少，表 2.1 列举了常用的 NPM 操作命令。

表 2.1 NPM 常用操作命令

命 令	说 明
npm install	安装模块
npm uninstall	卸载模块
npm update	更新模块

续表

命 令	说 明
npm outdated	检查模块是否已经过时
npm ls	查看安装的模块
npm init	在项目中引导创建一个package.json文件
npm help	查看某条命令的详细帮助
npm root	查看包的安装路径
npm config	管理npm的配置路径
npm cache	管理模块的缓存
npm start	启动模块
npm stop	停止模块
npm restart	重新启动模块
npm view	查看模块的注册信息
npm version	查看模块版本
npm test	测试模块
npm adduser	添加用户
npm publish	发布模块
npm access	在发布的包上设置访问级别

提示：由于国内网络环境限制，访问 NPM 时常无法连接或者访问速度缓慢，这里笔者推荐淘宝的 NPM 镜像，详细操作请参考 <https://npm.taobao.org/>。

接着，尝试使用 NPM 安装 Koa（基于 Node.js 平台的下一代 Web 开发框架），命令如下：

```
npm install koa
```

命令执行效果如图 2.9 所示。

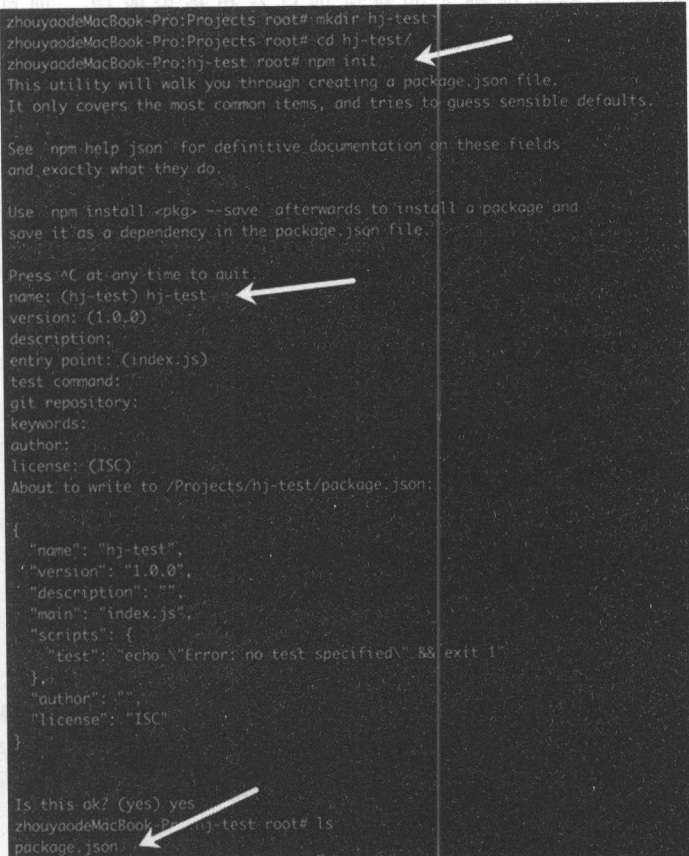
```
zhouyaodeMacBook-Pro:koa root# npm install koa
/Projects
-- koa@1.2.4
+++ accepts@1.3.3
| -- negotiator@0.6.1
+++ co@4.6.0
+++ composition@2.3.0
| -- any-promise@1.3.0
+++ content-disposition@0.5.2
+++ content-type@1.0.2
+++ cookies@0.6.2
| -- depd@1.1.0
| -- keygrip@1.0.1
+++ debug@2.5.2
| -- ms@0.7.2
```

图 2.9 使用 NPM 安装 Koa

NPM 上提供的包模块都是免费的，同样，如果自己开发了一个不错的 Node.js 应用或者模块，也可以发布到 NPM 站点上供所有人使用，步骤如下。

(1) 前往 <https://www.npmjs.com/signup> 注册一个账号。

(2) 进入要发布工程的目录，为了方便演示，这里新建一个“hj-test”的目录，并创建 package.json 文件，相关命令如图 2.10 所示。



```

zhouyaodeMacBook-Pro:Projects root# mkdir hj-test
zhouyaodeMacBook-Pro:Projects root# cd hj-test/
zhouyaodeMacBook-Pro:hj-test root# npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (hj-test) hj-test
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to /Projects/hj-test/package.json:

{
  "name": "hj-test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}

Is this ok? (yes) yes
zhouyaodeMacBook-Pro:hj-test root# ls
package.json
  
```

图 2.10 使用 NPM 创建项目

(3) 运行命令“npm adduser”，输入之前在 NPM 官网注册的账户和密码，如图 2.11 所示。

```

zhouyaodeMacBook-Pro:hj-test root# npm adduser
Username: zhouyao
Password:
Email: (this IS public) zhouyao@hujiang.com
Logged in as zhouyao on https://registry.npmjs.org/.
  
```

图 2.11 添加 NPM 账户

(4) 执行“npm publish”命令将测试项目发布到 NPM 站点，如图 2.12 所示。


```
zhouyaodeMacBook-Pro:~$ npm publish  
+ hj-test@1.0.0
```

图 2.12 发布到 NPM 站点

(5) 通过 NPM 官网验证刚才的测试项目发布是否成功, 项目搜索结果地址为 <https://www.npmjs.com/search?q=hj-test>, 已经发布成功, 如图 2.13 所示。

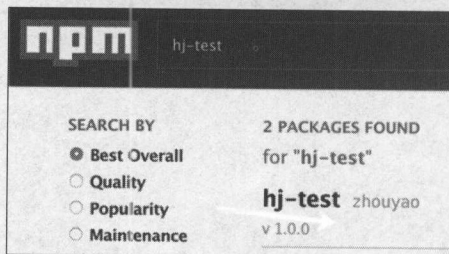


图 2.13 hj-test 测试项目发布成功

在使用 NPM 管理 Node.js 模块时, 涉及安装、升级和部署, 由于第三方依赖包的操作不当或者不遵守开发规范, 很可能出现项目依赖的包在不同阶段出现了不同的版本, 最终导致安装运行失败或者线上部署错误等问题。笔者推荐使用 NPM 的 `shrinkwrap` 命令来避免该问题。NPM `shrinkwrap` 会按照当前项目的 `node_modules` 目录生成稳定的依赖版本描述。简言之, 即使依赖包后续再有版本升级, 项目在不同时间节点的交付和部署都是当初开发完毕并且测试通过的状态。

提示: Facebook 在 2016 年推出了 Yarn, 一个快速、可靠和安全的 JavaScript 依赖管理工具。笔者目前也在使用 Yarn 来替代 NPM, 在依赖包的下载速度还有版本依赖管理上较之 NPM 均更加优秀, 项目地址为 <https://yarnpkg.com/>。

2.2.4 Web 代理工具 NProxy

Web 代理工具 Windows 系统上有 Fiddler, Mac OS 上有 Charles。其实还有一个非常方便的能够在多端上使用的工具, 就是接下来要介绍的 NProxy。

NProxy 项目发布于 2013 年, 官网地址为 <http://goddyzhao.me/nproxy/>, 特点如下:

- 支持 Mac、Linux 和 Windows。
- 同时支持单文件和 combo 文件替换。
- 支持目录替换。
- 同时支持 HTTP 和 HTTPS 两种协议。

使用 NPM 安装 NProxy，依赖的 Node.js 版本至少为 0.8.x，命令如下：

```
npm install -g nproxy
```

安装完毕后，执行命令获取当前 NProxy 版本号，如下：

```
nproxy -V
1.4.5 // 返回当前 NProxy 版本号
```

除了参数“-V”外，NProxy 还提供了其他的参数，如下：

- -h: --help，输出使用帮助信息。
- -l: --list [list]，定义替换规则文件。
- -p: -port [port]，定义代理监听端口，默认为 8989。
- -t: --timeout [timeout]，定义请求超时时间，默认为 5 秒。

项目提供了模板配置文件，读者可以根据项目实际情况进行调整，配置代码如下：

```
module.exports = [
  // 1.单文件替换（本地）
  {
    pattern: 'homepage.js', // 匹配替换的单个文件
    responder: "/home/goddyzhao/workspace/homepage.js"
  },
  // 2.单文件替换（在线）
  {
    pattern: 'homepage.js', // 匹配替换的单个文件
    responder: "http://www.anotherwebsite.com/assets/js/homepage2.js"
  },
  // 3.以本地文件绝对路径匹配线上 combo 文件
  {
    pattern: 'group/homepageTileFramework.*.js',
    responder: [
      '/home/goddyzhao/workspace/webapp/ui/homepage/js/a.js',
      '/home/goddyzhao/workspace/webapp/ui/homepage/js/b.js',
    ]
  },
  // 4.以本地指定目录和相对文件地址匹配线上文件
  {
    pattern: 'group/homepageTileFramework.*.js',
    responder: {
      dir: '/home/goddyzhao/workspace/webapp/ui/homepage/js',
      src: ['a.js', 'b.js']
    }
  }
]
```

```

    }
  },
  // 5.映射服务端到本地图片目录
  {
    pattern: 'ui/homepage/img',           // 类型字符串
    responder: '/home/goddyzhao/image/'    // 必须为绝对目录地址
  },
  // 6.通过正则表达式匹配请求，responder 可获得正则匹配变量，如$1、$2。
  {
    pattern: /https?:\/\/\[\\w\.\]*((?:\d+)?\/ui\/(.*?)_dev\.(\w+))\/,
    responder: 'http://localhost/proxy/$1.$2'
  },
  // 7.通过正则表达式映射服务端图片目录到本地图片目录
  // 下面的规则可以匹配多种路径到相应的本地文件
  // 匹配实例，如下：
  // http://host:port/ui/a/img/... => /home/a/image/...
  // http://host:port/ui/b/img/... => /home/b/image/...
  // ...
  {
    pattern: /ui\/(.*?)\/img\/\//,
    responder: '/home/$1/image/'
  }
];

```

将上述代码保存至文件“replace_rule.js”，启动 NProxy，命令如下：

```
nproxy -l replace_rule.js
```

服务规则生效后，还需要在浏览器中配置对应的代理信息。默认配置是，代理 IP 为 127.0.0.1，端口为 8989。

2.2.5 HTTP 服务器 http-server

http-server 是一个简单的零配置命令行 HTTP 服务器，非常适合日常的测试、本地开发等环境。本书中大部分实例均可以通过 http-server 进行部署访问。

使用 NPM 安装 http-server，命令如下：

```
npm install http-server -g
```

安装完毕后，可以直接进入项目文件目录启动 HTTP 服务，命令如下：

```
http-server
```

运行成功，读者可以访问 `http://localhost:8080` 查看服务。`http-server` 还提供了多种参数，命令语法格式如下：

```
http-server [path] [options]
```

- “[path]” 默认为当前项目的 `public` 文件夹，如果不存在则指定为当前根目录。
- “[options]” 参数说明见表 2.2。

表 2.2 “[options]” 参数说明

参 数 名	说 明
-p	端口号（默认为8080）
-a	IP地址（默认为0.0.0.0）
-d	显示目录列表（默认为true）
-i	是否在访问目录地址时，默认显示以“index”命名的网页文件（默认为true）
-g 或 --gzip	是否启动gzip（默认为false）
-e 或 --ext	默认显示文件的扩展名（默认为html）
-s 或 --silent	禁止日志信息输出
--cors	启动CORS通过Access-Control-Allow-Origin协议头
-o	启动服务后打开浏览器查看当前项目
-c	通过cache-control max-age协议头设置缓存时间，单位秒。例如，“-c10”表示缓存10秒（默认为3600秒）。关闭缓存使用“-c-1”
-U 或 --utc	在日志消息中使用UTC时间格式
-P 或 --proxy	代理所有不存在的请求地址至指定网址，比如：-p <code>http://www.hujiang.com</code>
-S 或 --ssl	启动HTTPS
-C 或 --cert	ssl证书文件地址（默认为cert.pem）
-K 或 --key	ssl密钥文件地址（默认为key.pem）
-r 或 --robots	提供一份爬虫协议（默认为所有收入，即“User-agent: * Disallow: /”）
-h 或 --help	打印帮助说明并退出

2.3 本章小结

本章首先给读者推荐了一款免费跨平台编辑器 Visual Studio Code。之后，着重介绍了 Node.js 和 NPM 的基本安装和使用，同时还介绍基于 Node.js 的 Web 代理工具 NProxy 和 HTTP 服务器 `http-server`。上述工具可以帮助读者学习使用本书中所提供的实例。本书的部分实例还需要依赖于 Java、Python、Ruby 等开发环境，本章不再赘述，读者可以参考其官网安装部署。

3

第 3 章

HTML 5 必会实际常用特性

HTML 作为 Web 开发的基石，从诞生至今，已经走到了第 5 个版本。2014 年 10 月，W3C 对外宣布 HTML 5 标准规范，至今已经过去 3 个年头。3 年里，各大厂商的浏览器纷纷支持了绝大部分 HTML 5 新特性，本章将从实际常用特性出发，向读者介绍如何使用 HTML 5。

3.1 新的语义

HTML 已经是所有网页的标配，但 HTML 5 提供了新的语义，本节就从新特色入手，介绍 HTML 5。

3.1.1 新元素的到来

HTML 5 出现之前，通常使用 DIV 元素结合样式类名来表述元素语义，此处语义的使用受益者集中在开发人员，开发者可以方便地通过该方式了解元素含义，减少维护开发成本。但语义本

身的意义,更关键在于让机器能够读懂页面的含义。从互联网诞生的第一天开始,“自由、开放、平等、分享”就作为核心思想促进其繁荣发展,让机器能够识别 HTML 结构中的语义含义也正是 HTML 5 新元素的本质作用。

开发者在 Web 页面中,经常会碰到包含头部、侧边栏和底部这样的布局,如图 3.1 所示。

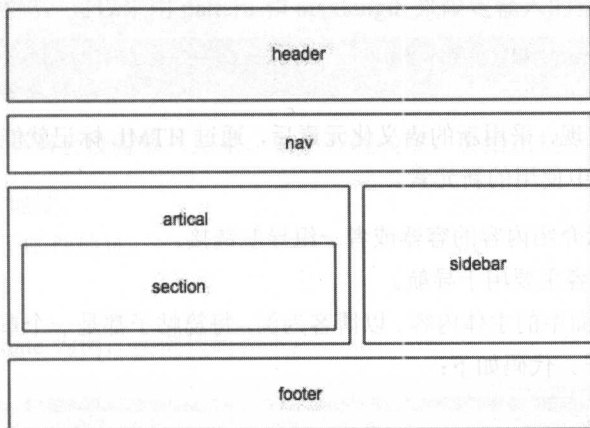


图 3.1 页面布局

图 3.1 所使用的布局采用 DIV 方式实现,代码如下:

```
<body>
  <div class="header"></div>          <!-- 页头-->
  <div class="nav"></div>              <!-- 导航-->
  <div class="main">                  <!-- 主体内容-->
    <div class="artical">             <!-- 文章-->
      <div class="section"></div>      <!-- 节-->
    </div>
    <div class="sidebar"></div>        <!-- 边栏-->
  </div>
  <div class="footer"></div>          <!-- 页脚-->
</body>
```

在上述代码中,为了便于阅读,需要采用 Class 样式类名来定义不同的内容块。可是搜索引擎并不能识别这些人为的自定义格式。接着,采用 HTML 5 提供的新元素重新实现上述布局,代码如下:

```
<body>
  <header></header>
```



```

<nav></nav>
<div>
  <article>
    <section></section>
  </article>
  <aside></aside>
</div>
<footer></footer>
</body>

```

和 DIV 实现对比发现，采用新的语义化元素后，通过 HTML 标记就能知道文档内容结构。接下来，分别介绍本实例中使用的新元素。

- **Header:** 可表示介绍内容的容器或者一组导航链接。
- **Nav:** 标签的内容主要用于导航。
- **Article:** 标识页面中的主体内容。以博客为例，每篇帖子都是一个重要内容，可采用 Article 标识每一个帖子。代码如下：

```

<article>
  <h2>post title</h2>
  <p>post contents</p>
</article>
<article>
  .....
</article>

```

- **Section:** 用来标记页面上重要的部分。该标记类似于将文档分为多个章节。
- **Aside:** 表示和页面主要内容相关，但不是页面的一部分，经常表示一个相关链接。
- **Footer:** 和 Header 相对，表示文档或者章节的页脚，比如版权等信息放在此标记中。

HTML 5 还提供了很多丰富的语义化标记，如 Address、Mark、Time 等标记，这里就不一一介绍了。

3.1.2 表单的增强应用

在 HTML 4 中，提供了一些简单的表单元素应对基本输入。对于特定类型的输入，如日期时间的输入，交互体验无法满足需求。因此，为了得到更好的效果，只能采用 JavaScript 编写组件来实现。在 HTML 5 中得到了完善，增加了新的表单元素来提供更多输入类型。本节将依次介绍这些新功能。

1. Input 元素的 Type 属性扩充

在 HTML 4 中, Input 元素只允许输入纯文本, 并且不具备类型。HTML 5 中对 Input 元素的 Type 属性进行了扩充, 比如:

- search: 呈现一个搜索框。
- tel: 输入电话号码, 可以采用 pattern 和 maxlength 来限定输入的格式。实例代码如下:

```
<input type="tel" name="tel" value="" placeholder="请输入手机号码" pattern="1[3-8][0-9]{9}" title="请输入 11 位手机号">
```

在提交表单时, 浏览器会校验输入结果。如果校验失败, 会给出提示信息, 如图 3.2 所示。

- url: 输入 URL 地址。
- email: 输入电子邮件地址。
- date: 输入日期。

以 Type 属性选择 date 为例, 实例代码如下:

```
<input type="date">
```

单击输入框, 效果如图 3.3 所示。

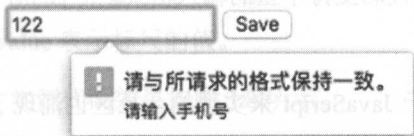


图 3.2 电话号码校验失败提示



图 3.3 日期选择效果

注意: 日期选择在不同的浏览器中实现的效果有所差异。图 3.3 是 Mac 系统下的 Chrome 浏览器效果。

- color: 输入颜色。

以 Type 属性选择 color 为例, 实例代码如下:

```
<input type="color">
```

单击颜色选择输入框, 效果如图 3.4 所示。

- **number:** 输入数字。
- **range:** 滑块输入。

以 Type 属性选择 range 为例, 实例代码如下:

```
<input type="range" min="20" max="100" step="2" >
```

输入框效果如图 3.5 所示。

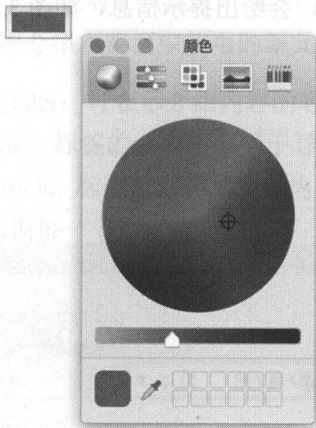


图 3.4 颜色选择效果



图 3.5 range 输入效果

Input 还提供了一些其他的输入类型, 如 `datetime`、`month`、`week` 等, 这里就不一一列举介绍了。通过对输入类型的这些扩展, 使开发者能使用规范的代码支持丰富的输入格式。

2. Input 通过属性进行表单验证

在 HTML 4 中, 对于表单输入做验证, 一般是借助于 JavaScript 来实现的。社区也涌现了各种各样的表单验证插件。在 HTML 5 中可以通过属性支持表单验证, 由 `required` 和 `pattern` 属性实现。

- **required:** 标记当前 Input 元素为必填, 代码如下:

```
<input type="text" placeholder="此项必填" required>
```

输入框效果如图 3.6 所示。

- **pattern:** 采用正则表达式验证表单输入。

提示：对于校验失败的提示信息，可以采用 title 属性指定。

3. Input 元素的其他有用属性

Input 元素还提供了如下一些有用的属性。

- autofocus：当页面加载时，自动聚焦到当前 Input 元素。
- form：将 Input 元素和特定的 Form 表单关联。
- placeholder：输入占位符，提示用户输入，效果如图 3.6 所示。

4. HTML 5 的新元素和特殊属性 contenteditable

HTML 5 除了增强了 Input 元素外，还提供了如下一些新的元素。

- progress 元素表示进度条，实例代码如下：

```
<progress value="30" max="100"></progress>
```

效果如图 3.7 所示。

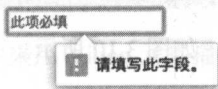


图 3.6 必填效果



图 3.7 进度条效果

- Meter 元素表示标尺，实例代码如下：

```
<meter value="3" min="0" max="10">3/10</meter><br>
<meter value="0.6">60%</meter>
```

Meter 元素包含三个属性：max 表示标尺的最大值，默认为 1；min 表示标尺最小值，默认为 0；value 表示标尺的值。

实例代码效果如图 3.8 所示。

- HTML 5 还提供了一个特殊属性 contenteditable，通过该属性，可以让一个普通的元素可编辑。如对段落元素设置了该属性后，相当于一个简单的编辑器，代码如下：

```
<p contenteditable="true">这里的内容是可以编辑的</p>
```

编辑效果如图 3.9 所示。

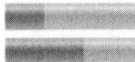


图 3.8 meter 元素效果图

这里的内容是可以编辑的

图 3.9 contenteditable 属性效果

HTML 5 提供的新特性，使得开发者能够在 HTML 层面处理各种类型的输入、验证、自动聚焦等功能。但有些可惜的是，各浏览器对一些展现样式有不同的实现，如果要统一风格，依旧需要采用传统的方式。

3.1.3 使用音频和视频

HTML 4 时代，在网页中播放多媒体视频和音频是一件比较困难的事情，需要借助浏览器插件来完成，比如通过 Flash 技术。直到 HTML 5 提供了 Audio 和 Video 标签，才可以很方便地在网页中嵌入音频和视频。

首先介绍 Audio 标签，它用于播放音频，实例代码如下：

```
<audio controls>
  <source src="vincent.ogg" />
  <source src="vincent.mp3" />
  你的浏览器不支持 Audio 标记
</audio>
```

在 Chrome 浏览器中运行上述代码，可以看到播放器界面，如图 3.10 所示。



图 3.10 音频播放界面

由于专利等各方面原因，各大浏览器对于音频格式的支持并不一致。可以在 Audio 元素内部指定多个播放源设置音频地址。这样，当浏览器不能识别上一个音频时，可以继续尝试下一个音频。当浏览器不能识别 Audio 元素时，可以将 Audio 元素内的提示文本信息呈现给用户，提示该功能不被浏览器支持。

Audio 元素控制行为的属性如下。

- **controls**: 该属性控制是否显示标准的音频空间。
- **autoplay**: 是否自动播放，默认为 false。
- **loop**: 是否循环播放，默认为 false。
- **preload**: 预先加载的方式。有三种情况：**none** 表示不预加载；**metadata** 表示只加载音频的元数据；**auto** 表示预加载整个音频。默认值为 auto。
- **volum**: 音量，值在 0 至 1 之间。

Audio 元素提供一系列 API 用于自定义播放界面和控制音频播放，实例代码如下：


```

<audio id="audio">
  <source src="vincent.ogg" />
  <source src="vincent.mp3" /> 你的浏览器不支持 Audio 标记
</audio>
<p>
  <button id="btnPlay">Play</button>
  <button id="btnPause">Pause</button>
</p>
<script>
  var audio = document.getElementById("audio")
  document.getElementById("btnPlay").addEventListener("click", function(){
    audio.play()
  })
  document.getElementById("btnPause").addEventListener("click", function(){
    audio.pause()
  })
</script>

```

在上述实例中,通过脚本获取 Audio 元素,调用元素对象的 play 和 pause 方法播放和暂停音频。

HTML 5 中采用 Video 元素播放视频,与 Audio 元素相似,各大浏览器支持的格式存在差异,实例代码如下:

```

<video width="400" height="300" controls>
  <source src="dizzy.mp4" type="video/mp4">
  <source src="dizzy.webm" type="video/webm">
  <source src="dizzy.ogv" type="video/ogg">
  <p>你的浏览器不支持 HTML 5 视频</p>
</video>

```

在 Chrome 浏览器中运行后,效果如图 3.11 所示。

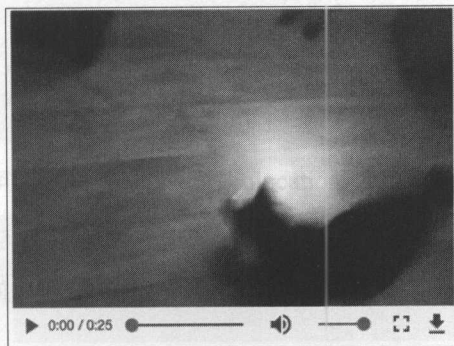


图 3.11 Video 播放效果

通过 API 控制 Video 的播放时间位置, 代码如下:

```
<input type="number" name="time" value="10" id="time">
<button id="btnSeek">GetInfo</button>
<script>
  var video = document.getElementById("video")
  var time = document.getElementById("time")
  document.getElementById("btnSeek").addEventListener("click", function () {
    video.currentTime = time.value;
  })
</script>
```

在上述代码中, 通过指定属性 `currentTime` 来控制播放的位置。也可以在 `source` 的 URL 上指定媒体的起始和结束时间。如控制视频从第 5 秒开始播放, 代码如下:

```
<source src="dizzy.mp4#t=5" type="video/mp4">
```

如果需要控制视频从 5 秒开始播放到 10 秒结束, 代码如下:

```
<source src="dizzy.mp4#t=5,10" type="video/mp4">
```

如果需要控制视频从头开始, 播放到第 15 秒结束, 代码如下:

```
<source src="dizzy.mp4#t=,15" type="video/mp4">
```

可以对最后一个 `source` 元素绑定 `error` 事件, 检测媒体元素是否存在可用源, 如果没有, 则向用户给出友好提示, 实例代码如下:

```
var sources = video.querySelectorAll("source")
var lastSource = sources[sources.length - 1]
lastSource.addEventListener("error", function(){
  alert('No source available')
})
```

本节简要介绍了 HTML 播放音频和视频的方式。对于视频播放, 实际场景中大多采用流媒体的方式, 限于篇幅将不再描述。

3.2 访问你的设备

对于移动开发来说, 很重要的功能就是访问智能设备, 本节介绍如何用 HTML 5 访问你的设备。

3.2.1 定位当前地理位置

根据用户的地理位置提供相关地区服务，已经是非常普遍的一项功能，例如本地生活服务类网站、外卖网站等。HTML 5 新功能中提供了获取用户位置的能力，本节将会介绍如何使用 HTML 5 Geolocation API 来构建基于地理位置的应用。

首先来了解一下各主要浏览器对 HTML 5 Geolocation 的支持情况，见表 3.1。

表 3.1 浏览器对HTML5 Geolocatio的支持

浏览器	版本		只支持HTTPS版本
IE	9+	-	
Edge	12+	-	
Firefox	3.5+	-	
Chrome	5+	50+	
Safari	5+	39+	
iOS Safari	3.2+	10.2+	
Android Browser	2.1+	56+	
Chrome for Android	57+	57+	
UC Browser fro Android	11.4+	-	

注意：出于安全考虑，部分最新的浏览器只允许通过 HTTPS 协议使用 Geolocation API。在 HTTP 协议下使用 Geolocation API，浏览器会抛出异常。在开发阶段，127.0.0.1 和 localhost 等本地域在两种协议下均可以使用。

Geolocation API 通过 navigator.geolocation 全局对象进行访问。初次访问时，浏览器会询问用户是否允许共享位置，若用户选择允许则程序获得使用 Geolocation API 权限。判断浏览器是否支持 Geolocation API，可以通过判断是否存在 navigator.geolocation 对象来得知，实例代码如下：

```
if (navigator.geolocation) {
    // 获取地理位置
} else {
    alert('您的浏览器不支持 Geolocatioin!');
}
```

获取用户的当前位置，可以调用 navigator.geolocation 的 getCurrentPosition 方法，格式如下：

```
navigator.geolocation.getCurrentPosition(
    function success(position){},
    function error(positionError){},
    options
)
```

实例代码如下:

```
function success(position){
    console.log('获取位置成功: ', position.coords);
}
function error(positionError){
    console.log('获取位置失败: ', positionError.code, positionError.message);
}
var options = {
    enableHighAccuracy: false,
    timeout: 30000,
    maximumAge: 0
};
navigator.geolocation.getCurrentPosition(success, error, options)
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(success, error, options);
} else {
    alert('您的浏览器不支持 Geolocation!')
}
```

成功获取位置后, 会调用回调函数 `success`。返回的参数 `position` 对象包含获取位置时的时间戳 `timestamp` 和坐标信息 `coords`。`coords` 对象包含了很多有用的位置数据信息, 列举几个常用的属性, 如下。

- `latitude`: 坐标纬度。
- `longitude`: 坐标经度。
- `accuracy`: 坐标精度, 单位为米。

当获取位置失败时, 会调用回调函数 `error`。返回的参数 `positionError` 的 `message` 属性包含了相关的错误信息描述, `positionError.code` 标识错误的原因, `positionError.code` 的值有以下几种。

- `UNKNOWN_ERROR(0)`: 其他错误。
- `PERMISSION_DENIED(1)`: 用户拒绝分享位置信息。
- `POSITION_UNAVAILABLE(2)`: 获取用户位置信息失败。
- `TIMEOUT(3)`: 获取用户位置信息超时。

`getCurrentPosition` 方法的参数 `options` 可以用来设置以下内容。

- `enableHighAccuracy`: 布尔值, 是否获取高精度的位置信息, 如果开启, 可能会增加响应时间, 默认值为 `false`。

- **timeout**: 定位超时时间, 单位毫秒, 如到达时间时没有取得用户位置信息, 则触发失败回调函数, 默认值为 0, 表示无限大。
- **maximumAge**: 用户位置信息缓存的最大时间 (单位为毫秒), 默认值为 0。

当用户位置变化时, 还可以通过 `watchPosition` 方法监听用户的位置信息, `watchPosition` 的参数和 `getCurrentPosition` 相同。函数执行后返回一个唯一标识, 可以使用 `clearWatch` 方法来取消监听, 实例代码如下:

```
var watchId = navigator.geolocation.watchPosition(success, error, options);
navigator.geolocation.clearWatch(watchId);
```

3.2.2 实战演练: 调用摄像头拍个照

HTML 5 的 `getUserMedia` API 提供了访问用户媒体设备的能力, 基于该特性, 开发者可以在不依赖任何浏览器插件的条件下访问视频和音频等设备。本节会介绍 HTML 5 `getUserMedia` API 的使用方法, 并完成一个调用摄像头实现拍照功能的实例。

1. `getUserMedia` API

`getUserMedia` API 起初的版本是 `navigator.getUserMedia`, 目前已从最新 Web 标准中废除, 一部分浏览器仍然支持, 但将来有可能被废弃。

最新的标准 `getUserMedia` API 为 `navigator.mediaDevices.getUserMedia`, 但浏览器支持情况不如旧版 API 普及。

各主要浏览器对 `getUserMedia` API 的支持情况见表 3.2。

表 3.2 浏览器对 `getUserMedia` API 的支持情况

浏览器	版本 (旧版API)	版本 (新版API)	说明
IE	不支持	不支持	
Edge	12+	不支持	
Firefox	17+	36+	需要前缀moz
Chrome	21+	47+	部分支持, 需要前缀webkit
Safari	不支持	不支持	
iOS Safari	不支持	不支持	
Android Browser	56+	不支持	部分支持, 需要前缀webkit
Chrome for Android	57+	不支持	部分支持, 需要前缀webkit
UC Browser fro Android	11.4+	不支持	部分支持, 需要前缀webkit

旧版 Geolocation API 可以通过 `navigator.getUserMedia` 来调用, 但是各浏览器对 `getUserMedia`

的支持有很大差异, 部分浏览器需要使用特定前缀, 例如在 Firefox 中, 方法为 navigator.mozGetUserMedia。在使用之前, 需要确定浏览器是否支持 getUserMedia, 判断方法如下:

```
if (navigator.mediaDevices.getUserMedia || navigator.getUserMedia ||
    navigator.webkitGetUserMedia || navigator.mozGetUserMedia) {
  // 调用用户媒体设备
} else {
  alert('您的浏览器不支持访问用户媒体设备!');
}
```

旧版 getUserMedia 语法如下:

```
getUserMedia(constraints, successCallback, errorCallback)
```

新版 getUserMedia 语法如下:

```
getUserMedia(constraints).then(successCallback).catch(errorCallback)
```

- 参数 constraints, 指定请求的媒体类型, 主要包含 video 和 audio, 例如请求不带任何参数的视频和音频, 代码如下:

```
{ video: true, audio: true }
```

可指定视频分辨率, 代码如下:

```
{ video:{ width: 640, height: 360 } }
```

移动设备上, 可指定使用前置摄像头, 代码如下:

```
{ video:{ facingMode: 'user' } }
```

或者使用后置摄像头, 代码如下:

```
video:{ facingMode: { exact: 'environment' } } }
```

- 成功回调函数 successCallback 的参数 stream, 为 MediaStream 对象, 表示媒体内容的数据流, 可以通过 URL.createObjectURL 转换后设置为 Video 或 Audio 元素的 src 属性来使用, 部分较新的浏览器也可以直接设置为 srcObject 属性来使用。
- 失败回调函数 errorCallback 的参数 error, 其中 name 属性的值参考表 3.3。

表 3.3 错误值及名称

错误值	错误名称
AbortError	中止错误
NotAllowedError	拒绝错误
NotFoundError	找不到错误

续表

错 误 值	错误名称
NotReadableError	无法读取错误
OverConstrainedError	无法满足要求错误
SecurityError	安全错误
TypeError	类型错误

`navigator.mediaDevices.getUserMedia` 使用实例代码如下：

```
navigator.mediaDevices.getUserMedia({ video: true })
  .then(function(stream) {
    // 成功获得媒体流
  })
  .catch(function(error) {
    console.log('获取用户媒体失败: ' + error.name);
  });
```

初次访问用户媒体设备时，浏览器会询问用户是否允许访问，在用户允许后获得访问媒体设备授权。

2. 调用摄像头拍照实例

本例子中，将请求访问用户摄像头，并把视频流通过 `Video` 元素显示出来。实例中提供一个“拍照”按钮，通过 `Canvas` 将 `Video` 的画面截取并绘制。实例的 HTML 部分代码如下：

```
<!-- video 用于显示媒体设备的视频流，自动播放 -->
<video id="video" autoplay style="width:480px;height:320px"></video>
<div><button id="capture">拍照</button></div>           <!-- 拍照按钮 -->
<canvas id="canvas" width="480" height="320"></canvas>    <!-- 描绘 video 截图 -->
```

JavaScript 代码如下：

```
01 // 访问用户媒体设备的兼容方法
02 function getUserMedia(constraints, success, error) {
03     if (navigator.mediaDevices.getUserMedia) {
04         // 最新的标准 API
05         navigator.mediaDevices.getUserMedia(constraints).then(success).catch(error);
06     } else if (navigator.webkitGetUserMedia) {
07         // Webkit 核心浏览器
08         navigator.webkitGetUserMedia(constraints, success, error);
09     } else if (navigator.mozGetUserMedia) {
10         // Firefox 浏览器
11         navigator.mozGetUserMedia(constraints, success, error);
```



```

12     } else if (navigator.getUserMedia) {
13         // 旧版 API
14         navigator.getUserMedia(constraints, success, error);
15     }
16 }
17 var video = document.getElementById("video");           // video 元素
18 var canvas = document.getElementById("canvas");         // canvas 元素
19 var context = canvas.getContext("2d");
20
21 // 成功的回调函数
22 function success(stream) {
23     // 兼容的 webkit 核心浏览器
24     var CompatibleURL = window.URL || window.webkitURL
25     // 将视频流设置为 video 元素的源
26     video.src = CompatibleURL.createObjectURL(stream);
27     video.play();                                         // 播放视频
28 }
29
30 // 异常的回调函数
31 function error(error) {
32     console.log('访问用户媒体设备失败: ', error.name, error.message);
33 }
34
35 if (navigator.mediaDevices.getUserMedia || navigator.getUserMedia ||
36     navigator.webkitGetUserMedia || navigator.mozGetUserMedia) {
37     // 调用用户媒体设备, 访问摄像头
38     getUserMedia({ video: { width: 480, height: 320 } }, success, error);
39 } else {
40     alert('您的浏览器不支持访问用户媒体设备! ');
41 }
42
43 // 绑定拍照按钮的单击事件
44 document.getElementById("capture").addEventListener("click", function () {
45     context.drawImage(video, 0, 0, 480, 320);           // 将 video 画面在 canvas 上绘制出来
46 });

```

运行的效果如图 3.12 所示。图 3.12 的上半部分为摄像头实时拍摄的视频, 下半部分为用户单击“拍照”按钮后, 从视频中的截图。读者如果感兴趣可以对本实例进行功能加强, 比如使用 CSS 3 的滤镜实现模糊、黑白等效果。

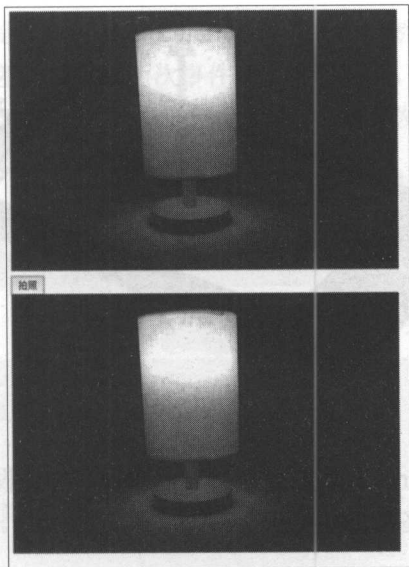


图 3.12 调用摄像头拍照

3.2.3 实战演练：在手机上实现摇一摇

现代手机都内置了方向传感器和运动传感器。通过传感器，可以感知手机的方向和位置的变化，基于此，可以开发出很多有趣的功能，如指南针、通过倾斜手机来控制方向的赛车游戏、甚至更热门的增强现实游戏等。HTML 5 提供了访问传感器信息的 API，分别是 DeviceOrientationEvent 和 DeviceMotionEvent，本节将介绍 API 的使用及通过 DeviceMotionEvent 实现摇一摇功能。

1. 方向事件和移动事件

在介绍这两个事件之前，需要先了解设备的方向变化和位置变化相关的概念。如图 3.13 标识了移动设备的三个方向轴。

如图 3.13 所示，X 轴表示左右横贯手机的轴，当手机绕 X 轴旋转时，移动的方向称为 Beta；Y 轴表示上下纵贯手机的轴，当手机绕 Y 轴旋转时，移动的方向称为 Gamma；Z 轴表示垂直于手机平面的轴，当手机绕 Z 轴旋转时，移动的方向称为 Alpha。

- 方向事件 deviceorientation

deviceorientation 事件是在设备方向发生变化时触发，使用方法如下：

```
window.addEventListener('deviceorientation', orientationHandler, true);
```

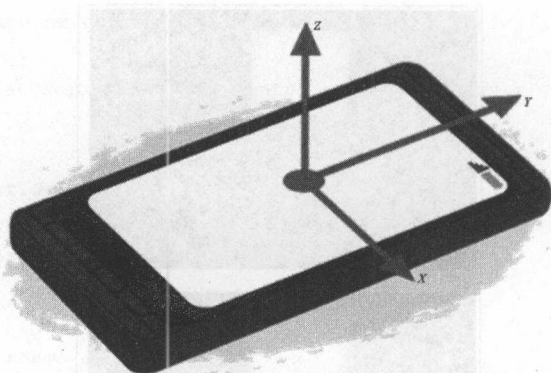


图 3.13 移动设备的三个方向的轴

回调函数 `orientationHandler` 在注册后，会被定时调用，并会收到一个 `DeviceOrientationEvent` 类型参数，通过该参数获取设备的方向信息，包含的属性见表 3.4。

表 3.4 `DeviceOrientationEvent`属性信息

属 性 名	说 明
<code>absolute</code>	如果方向数据跟地球坐标系和设备坐标系有差异，则为 <code>true</code> ，如果方向数据由设备本身的坐标系提供，则为 <code>false</code>
<code>alpha</code>	设备在Alpha方向上旋转的角度，范围为0~360度
<code>beta</code>	设备在Beta方向上旋转的角度，范围为-180~180度
<code>gamma</code>	设备在Gamma方向上旋转的角度，范围为-90~90度

• 移动事件 `devicemotion`

`devicemotion` 事件在设备位置发生变化时触发，使用方法如下：

```
window.addEventListener('devicemotion', motionHandler, false);
```

回调函数 `motionHandle` 在注册之后，会被定时调用，并会收到一个 `DeviceMotionEvent` 类型参数，通过该参数可以访问设备的方向和位置信息，包含属性信息见表 3.5。

表 3.5 `DeviceMotionEvent`属性信息

属 性 名	说 明
<code>acceleration</code>	设备在X、Y、Z三个轴的方向上移动的距离，已抵消重力加速
<code>accelerationIncludingGravity</code>	设备在X、Y、Z三个轴的方向上移动的距离，包含重力加速
<code>rotationRate</code>	设备在Alpha、Beta、Gamma三个方向旋转的角度
<code>interval</code>	从设备获取数据的频率，单位是毫秒

2. 摇一摇实例

摇一摇这个动作是指手机在一定时间内移动了一定距离，也可以近似地理解为手机以不低于

一定的速度移动。实现的方法是在监听 `devicemotion` 事件后,判断设备在 X、Y、Z 三个方向上移动的距离与前一次移动的距离差,并除以两次事件触发的时间差,即为设备移动的速度。将这个速度与预先设定的速度比较,如果大于预先设定值,则认为设备发生摇动。这个预先设定的速度可以进行多次测试之后,取其平均值。

摇一摇实例 HTML 代码如下:

```
<div>用力摇一摇你的手机</div>
```

JavaScript 代码如下:

```
01 var SHAKE_SPEED_THRESHOLD = 300; // 摇动速度阈值
02 var lastTime = 0; // 上次变化的时间
03 var x = y = z = lastX = lastY = lastZ = 0; // 位置变量初始化
04 function motionHandler(evt) {
05     // 取得包含重力加速的位置信息
06     var acceleration = evt.accelerationIncludingGravity;
07     var curTime = Date.now(); // 取得当前时间
08     if ((curTime - lastTime) > 120) { // 判断
09         var diffTime = curTime - lastTime; // 两次变化时间差
10         lastTime = curTime; // 保存此次变化的时间
11         x = acceleration.x;
12         y = acceleration.y;
13         z = acceleration.z;
14         // 计算摇动的速度
15         var speed = Math.abs(x + y + z - lastX - lastY - lastZ) / diffTime * 1000;
16         if (speed > SHAKE_SPEED_THRESHOLD) { // 速度是否大于预设速度
17             alert("你摇动了手机");
18         }
19         lastX = x; // 保存此次变化的位置 x
20         lastY = y; // 保存此次变化的位置 y
21         lastZ = z; // 保存此次变化的位置 z
22     }
23 }
24 if (window.DeviceMotionEvent) {
25     window.addEventListener('devicemotion', motionHandler, false);
26 } else {
27     alert('您的设备不支持位置感应');
28 }
```

在手机上访问该实例页面,快速摇动手机弹出提示,如图 3.14 所示。

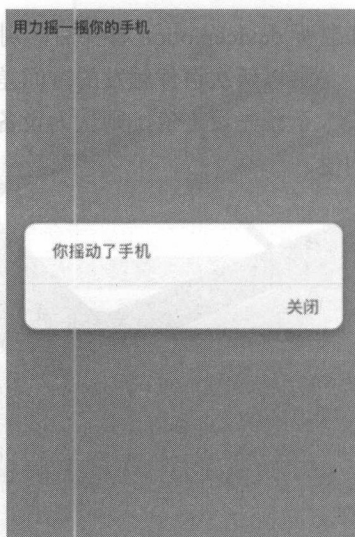


图 3.14 摇一摇运行结果

3.3 离线和存储

HTML 5 引入了应用程序缓存，这意味着 Web 应用可在没有网络时进行访问。同时，HTML 5 还提供了一套本地存储机制，允许开发者在本地存储少量数据，来提高用户体验。

3.3.1 实战演练：搭建一个简单的离线应用

离线应用是 HTML 5 新增的一项功能，旨在帮助用户在没有网络的情况下也可以使用 Web 应用程序。HTML 5 离线功能包含离线资源缓存、在线状态监测、本地数据存储等方面的内容。

- 离线资源缓存：通过浏览器机制将在线资源缓存到本地，当用户离线访问应用程序时，这些资源文件自动从本地加载，从而让用户可以正常使用应用程序。
- 在线状态监测：有些应用需要跟服务器做一些数据的交互，应用开发者需要知道浏览器是否处于在线状态，HTML 5 中提供了在线状态的监测。
- 本地数据存储：当应用程序处于离线状态时，程序需要把用户产生的数据存储到本地，以便在线时同步到服务器上。为此，HTML 5 提供了多种本地存储机制。

离线 Web 应用对比普通的 Web 应用多了一个描述文件，该文件用来列出需要缓存和永不缓存的资源，以备离线时使用。描述文件扩展名为“.manifest”或“.appcache”，推荐使用后者。描

述文件的 `mime-type` 类型为“`text/cache-manifest`”，且必须使用 UTF-8 编码。下面来看一个典型的描述文件内容，代码如下：

```
01 CACHE MANIFEST
02 #cache 之后的资源都会被缓存
03 CACHE:
04 main.html
05 style.css
06 main.js
07 #network 之后的资源不会被缓存，总是从线上获取
08 NETWORK:
09 account/
```

描述文件第一行必须是“`CACHE MANIFEST`”，以表明这是一个离线应用描述文件。文件中凡是以“`#`”开头的语句，都不会被解析，表示这是一句代码注释。

第3行代码“`CACHE:`”，这是一条指令，告诉浏览器在此之后的资源都将被离线存储。

第8行代码“`NETWORK:`”该指令与第3行的指令正好相反，表示在此之后的资源都不被离线存储，每次都从线上获取。

第9行为应用地址相对路径，此处表示在 `account` 目录下的所有资源都不用缓存。

假如开发一个“随手记”的 Web 应用，入口文件是 `main.html`，页面代码如下：

```
01 <html>
02 <head>
03 <meta charset="utf-8" />
04 <title>随手记</title>
05 <link rel="stylesheet" href="style.css" />
06 </head>
07 <body>
08 <h2>随手记--实时保存</h2>
09 <div>
10     <textarea id="content" cols="100" rows="20"></textarea>
11 </div>
12 <script src="main.js"></script>
13 </body>
14 </html>
```

业务逻辑 `main.js` 的代码如下：

```
01 // 获取记录内容的文本域
```



```

02 var el = document.querySelector('#content');
03 // 为文本域 DOM 节点添加 blur 事件
04 el.addEventListener('blur', function(){
05     // 获取文本域的内容
06     var data = el.value;
07     // 将内容保存到服务器
08     saveOnline(data);
09 });
10 // 保存内容的具体代码
11 function saveOnline(data){
12     var xhr = new XMLHttpRequest();
13     xhr.open('POST', '/savedata');
14     xhr.send('data='+data);
15 }

```

现在要将这个 Web 应用“离线”化，只需将 main.html 和 manifest 描述文件关联起来即可，关联代码如下：

```
<html manifest="./offline.appcache">
```

在 HTML 标签上添加 manifest 属性之后，浏览器就会自动下载 offline.appcache 文件，并进行解析，然后缓存文件中指定的资源。下次打开这个页面时，即便没有网络，也可以正常使用。但是，用户在离线状态下产生的数据并没有保存，所以为了保证离线状态下数据完整，还需要修改 JavaScript 代码，做数据的本地存储，修改后的 main.js 代码如下：

```

01 // 获取记录内容的文本域
02 var el = document.querySelector('#content');
03 // 为文本域 DOM 节点添加 blur 事件
04 el.addEventListener('blur', function(){
05     // 获取文本域的内容
06     var data = el.value;
07     // 如果是在线状态，就直接保存到服务器
08     if(navigator.onLine){
09         saveOnline(data);
10     }else{
11         // 如果是离线状态，则保存到本地
12         localStorage.setItem('data', data);
13     }
14 });
15 // 监听上线事件
16 window.online = function(){
17     // 从本地存储获取数据

```

```

18 var data = localStorage.getItem('data');
19 if (!!data){
20     // 如果数据存在，则保存到服务器
21     saveOnline(data);
22     // 同时，清空本地的存储
23     localStorage.removeItem('data');
24 }
25 };
26 // 保存内容的具体代码
27 function saveOnline(data){
28     var xhr = new XMLHttpRequest();
29     xhr.open('POST', '/savedata');
30     xhr.send('data='+data);
31 }

```

经过以上代码处理之后，“随手记”应用就算完美实现离线功能了，无论用户设备是否存在网络，都可以正常使用。

提醒：代码中的 `LocalStorage` 对象将在接下来的章节中会向大家详细介绍。

3.3.2 离线之后资源该如何更新——Service Worker

对于以上案例，细心的读者可能已经发现一个问题“离线之后资源该如何更新”，这也是每一个开发者都会考虑到的问题。HTML 5 提供了另外一套 API，帮助开发者完全控制离线数据，以支持更好的离线体验。该技术 W3C 在 2014 年提出，名为 `Service Worker`。

在 `Service Worker` 提出之前，HTML 5 通过 `Application Cache` 控制缓存，不过 `Application Cache` 有一些局限性，已经不被 W3C 推荐使用。

提示：`Application Cache` 技术细节可在线参考文档 https://www.w3schools.com/html/html5_app_cache.asp。

`Service Worker` 主要提供 4 类功能：

- 后台消息传递
- 网络代理
- 离线缓存
- 消息推送

`Service Worker` 是一段运行在浏览器后台进程脚本，独立于当前页面，并且不会直接参与

DOM 的操作。但是可以通过 `postMessage` 与页面通信, 页面根据 `postMessage` 传递的消息与 DOM 进行交互。下面来看一段 Service Worker 的实例代码:

```
01 // 注册 service worker
02 navigator.serviceWorker.register('sw.js').then(function(registration) {
03     console.log('Service Worker 注册成功');
04 }).catch(function (err) {
05     console.log('Servcie Worker 注册失败')
06 });
07 // sw.js 代码如下
08 // 需要缓存的资源列表
09 var cacheFiles = [
10     'style.css',
11     'main.js'
12 ];
13 // 在 install 事件里缓存资源
14 self.addEventListener('install', function (evt) {
15     evt.waitUntil(
16         caches.open('mycache').then(function (cache) {
17             return cache.addAll(cacheFiles);
18         })
19     );
20 });
```

这段代码通过完全手动的方式实现了对两个静态资源的缓存, 当然, 开发者也可以通过与服务器的通信决定何时缓存和更新。这种更灵活、更可控的开发模式, 也正是 Service Worker 被推崇的主要原因。

提示: 有关 Service Worker 的更多介绍, 请参考官方文档 <https://www.w3.org/TR/service-workers/>。

3.3.3 LocalStorage 与 SessionStorage

在 HTML 5 之前, Web 应用程序通用的数据存储方案一般通过 Cookie 实现。不过, 将数据存储在 Cookie 中有如下弊端:

- 大小受限, 标准浏览器下单个 Cookie 允许的大小是 4KB。
- 消耗性能, 当前域下的所有 HTTP 请求都会携带这些 Cookie 数据。

为了解决这些问题, HTML 5 提供了两种在客户端存储数据的新方法: LocalStorage 和 SessionStorage。

HTML 5 的本地存储为每个网站分配的空间大小是 5MB，虽然 5MB 看上去挺小，但对于文本存储来说，已能够满足业务大部分需求。

在 HTML 5 中，LocalStorage 与 SessionStorage 两种 API 在使用上没有区别，不过前者会一直存储在本地，直到手动清除，而后者则存活在当前页面的生命周期中，一旦页面关闭，存储的数据也自动消失。

本地存储的用法如下：

```
// 存储数据
localStorage.setItem('key', '需要存储的数据');
// 获取数据
var value = localStorage.getItem('key');
// 删除数据
localStorage.removeItem('key');
```

拿上一节中的“随手记”应用程序来举例，将数据存储到本地，确保下次打开时数据仍然存在，实例代码如下：

```
01 <html>
02 <head>
03 <meta charset="utf-8" />
04 <title>随手记</title>
05 </head>
06 <body>
07 <h2>随手记--本地保存</h2>
08 <div>
09     <textarea id="content" cols="100" rows="20"></textarea>
10 </div>
11 </body>
12 </html>
13 <script type="text/javascript">
14 // 获取记录内容的文本域
15 var el = document.querySelector('#content');
16 // 页面载入时，从本地获取存储的数据
17 el.value = localStorage.getItem('data') || '';
18 // 为文本域 DOM 节点添加 blur 事件
19 el.addEventListener('blur', function(){
20     // 获取文本域的内容
21     var data = el.value;
22     // 保存到本地
23     localStorage.setItem('data', data);
```



```

24  });
25  </script>

```

程序大致设计思路是使用 `LocalStorage` 进行存储，而当用户再次打开该应用程序时，将 `LocalStorage` 中存储的数据取出，并填充文本框。对于本地存储的数据，读者可以通过浏览器开发者工具查看，如图 3.15 所示。

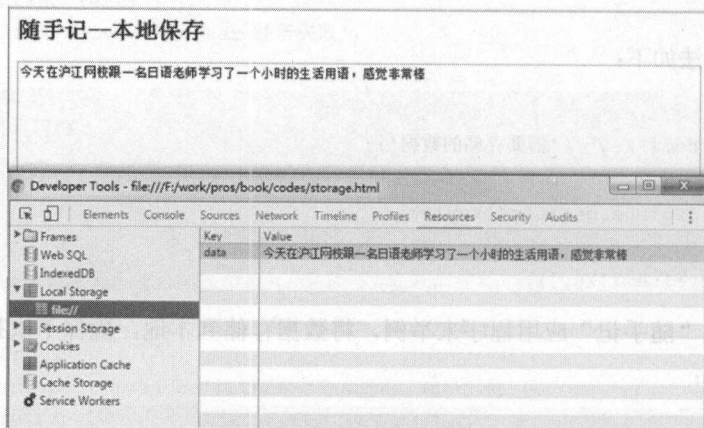


图 3.15 本地存储数据截图

本地存储除了上面提到的 `getItem` 和 `setItem` 方法，还可以使用索引和属性的形式来操作，代码如下：

```

// 存储数据
localStorage['key1'] = 'value1';
localStorage.key2 = 'value2';
// 获取数据
var value1 = localStorage['key1'];
var value2 = localStorage.key2;

```

另外本地存储也支持遍历和清空这些常用的方法，代码如下：

```

// 遍历所有存储的数据
for(var i=0;i<localStorage.length;i++){
    var key = localStorage.key(i);           // 获取 key
    var value = localStorage.getItem(key);    // 获取 value
    console.log(key, value);                 // 打印到控制台
}
localStorage.clear();// 清空所有数据

```


注意：本地存储只支持存储字符串类型数据，若要存储 JSON 数据，需要先将数据转换成字符串。

3.3.4 实战演练：利用 IndexedDB 实现便签管理

IndexedDB 是一个事务型数据库系统，同时也是一个基于 JavaScript 的面向对象的数据库系统。IndexedDB 为开发者在本地存储信息提供了另外一种可能性。与之前提到的 LocalStorage 和 SessionStorage 不同的是，IndexedDB 可以存储大量结构化的数据，并且使用基于索引的高效 API 检索。

本节将通过一个便签管理的实例，向读者演示如何使用 IndexedDB 存储数据。便签管理的页面代码如下：

```

01 <html>
02 <head>
03 <meta charset="utf-8" />
04 <title>indexedDB 使用实例</title>
05 <link rel="stylesheet" href="indexeddb.css" />
06 </head>
07 <body>
08 <!--创建一个便签容器-->
09 <div class="notes">
10     <!--添加按钮-->
11     <div class="add">
12         <p class="ic_add">+</p>
13         <p>添加便签</p>
14     </div>
15 </div>
16 <!--为了简化代码，基于 jQuery 开发-->
17 <script src="https://cdn.bootcss.com/jquery/3.2.1/jquery.min.js"></script>
18 <!--操作 indexedDB 的帮助类-->
19 <script src="indexeddb.js"></script>
20 <script>
21     // 预先定义每一个便签的 HTML 代码
22     var divstr = '<div class="note"><a class="close">X</a><textarea>
</textarea></div>';
23     // 实例化一个便签数据库、数据表
24     var db = new LocalDB('db1', 'notes');
25     // 打开数据库
26     db.open(function(){

```

```

27      // 页面初始化时, 获取所有已有便签
28      db.getAll(function(data){
29          var div = $(divstr);
30          div.data('id', data.id);
31          div.find('textarea').val(data.content);
32          // 将便签插入添加按钮前边
33          div.insertBefore(add);
34      });
35  });
36  // 为添加按钮注册单击事件
37  var add = $('#add').on('click', function(){
38      var div = $(divstr);
39      div.insertBefore(add);
40      // 添加一条空数据到数据库
41      db.set({content:''}, function(id){
42          // 将数据库生成的自增 id 赋值到便签上
43          div.data('id', id);
44      });
45  });
46  // 监听所有便签编辑域的焦点事件
47  $('#notes').on('blur', 'textarea', function(){
48      var div = $(this).parent();
49      // 获取该便签的 id 和内容
50      var data = { id: div.data('id'), content: $(this).val() };
51      // 写入数据库
52      db.set(data);
53  });
54  // 监听所有关闭按钮的单击事件
55  .on('click', '.close', function(){
56      if(confirm('确定删除此便签吗? ')){
57          var div = $(this).parent();
58          // 删除这条便签数据
59          db.remove(div.data('id'));
60          // 删除便签 DOM 元素
61          div.remove();
62      }
63  });
64 </script>
65 </body>
66 </html>

```

这个页面的核心 HTML 代码是一个便签容器和一个添加按钮, 页面加载后通过读取数据库现

有数据渲染便签列表。然后可以通过添加按钮添加新的便签,也可以通过删除按钮删除已有便签。页面运行效果如图 3.16 所示。

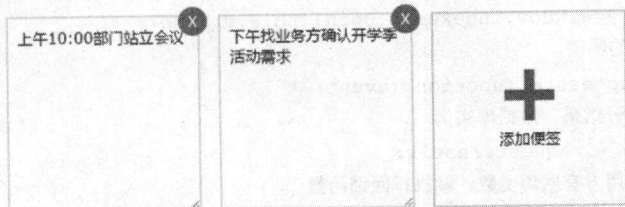


图 3.16 便签管理页面运行效果

同时,可以利用浏览器的开发者工具,查看保存到数据库的数据。首先打开浏览器的开发者工具(Developer Tools),然后切换到 Resources 选项卡,这时左侧出现了一系列本地存储的菜单。展开 IndexedDB 菜单,找到实例中创建的数据库“db1”,并单击数据表“notes”。此时右边区域出现该表的所有数据,查看结果如图 3.17 所示。

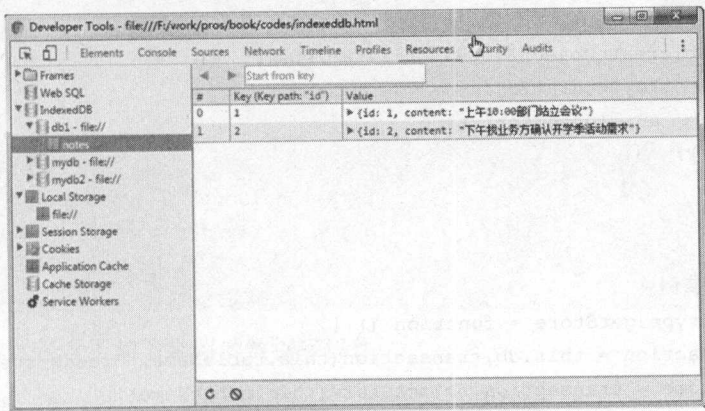


图 3.17 开发者工具中查看数据库截图

为了便于维护,实例中对 IndexedDB 操作的逻辑都封装在独立的 JavaScript 文件中,全部代码如下:

```
01 // 声明一个数据库操作的构造函数
02 function LocalDB(dbName, tableName) {
03     this.dbName = dbName;
04     this.tableName = tableName;
05     this.db = null;
06 }
07 // 在原型链上注册 open 方法,完成打开数据库的操作
```



```
08 LocalDB.prototype.open = function (callback) {
09     var _this = this;
10     // 执行打开数据库的动作
11     var request = window.indexedDB.open(_this.dbName);
12     // 打开成功后的回调
13     request.onsuccess = function (event) {
14         // 获取打开结果: 数据库实例
15         _this.db = request.result;
16         // 如果调用方有回调函数, 就执行回调函数
17         callback && callback();
18     };
19     // 第一次创建数据库时触发该事件
20     request.onupgradeneeded = function (event) {
21         // 获取数据库实例
22         var db = request.result;
23         // 检查是否存在指定的表
24         if (!db.objectStoreNames.contains(_this.tableName)) {
25             // 如果不存在, 则创建, 并指定一个自增的 id 作为查询依据
26             db.createObjectStore(_this.tableName, {
27                 keyPath: "id",
28                 autoIncrement: true
29             });
30         }
31     };
32 }
33 // 获取数据表的实例
34 LocalDB.prototype.getStore = function () {
35     var transaction = this.db.transaction(this.tableName, 'readwrite');
36     var objStore = transaction.objectStore(this.tableName);
37     return objStore;
38 }
39 // 保存一条数据: 支持添加和修改
40 LocalDB.prototype.set = function (data, callback) {
41     var objStore = this.getStore();
42     var request = data.id ? objStore.put(data) : objStore.add(data);
43     request.onsuccess = function (event) {
44         callback && callback(event.target.result);
45     };
46 }
47 // 获取一条数据
48 LocalDB.prototype.get = function (id, callback) {
```

```

49     var objStore = this.getStore();
50     var request = objStore.get(id);
51     request.onsuccess = function (event) {
52         callback && callback(event.target.result);
53     }
54 }
55 // 获取表中所有的数据
56 LocalDB.prototype.getAll = function (callback) {
57     var objStore = this.getStore();
58     // 打开数据游标
59     var request = objStore.openCursor();
60     request.onsuccess = function (event) {
61         var cursor = event.target.result;
62         if (cursor) {
63             // 如果游标存在, 执行回调并传入当前数据行
64             callback && callback(cursor.value);
65             // 继续下一行数据
66             cursor.continue();
67         }
68     }
69 }
70 // 删除一条数据
71 LocalDB.prototype.remove = function (id) {
72     var objStore = this.getStore();
73     objStore.delete(id);
74 }

```

以上代码基本上包含了如下常用的数据库操作。

- 打开数据库：如果指定的数据库不存在，系统会自动创建。
- 创建数据表：在首次打开数据库时检测表是否存在，不存在则创建。
- 保存一条数据：通过数据表对象的 `put` 和 `add` 方法实现。
- 获取一条数据：利用索引获取指定 `id` 的数据行。
- 获取所有数据：使用游标机制遍历所有的数据行。
- 删除一条数据：可以直接通过数据表对象的 `delete` 方法删除一条数据。

注意：IndexedDB 所有 API 操作均为异步模式，需要通过回调函数来获取结果。虽然规范里定义了 API 的同步版本，但是目前并没有在任何浏览器中得以实现。

3.4 图像效果

HTML 5 引入了 Canvas 和 SVG 标签为浏览器提供了更加丰富的图形渲染功能, 而 WebGL 用于在任何兼容的 Web 浏览器中呈现交互式 3D 和 2D 图形。本节将从基本的例子入手, 逐一介绍 Canvas、SVG 以及 WebGL 在浏览器图形处理中的运用。

3.4.1 使用 Canvas 绘制一个简单的饼图

Canvas 是 HTML 5 提供的新标签, 通过 JavaScript 可以在 Canvas 元素上绘制图形并实现动画效果。下面来看一下使用 Canvas 绘制一个简单饼图的基本过程, 如图 3.18 所示。

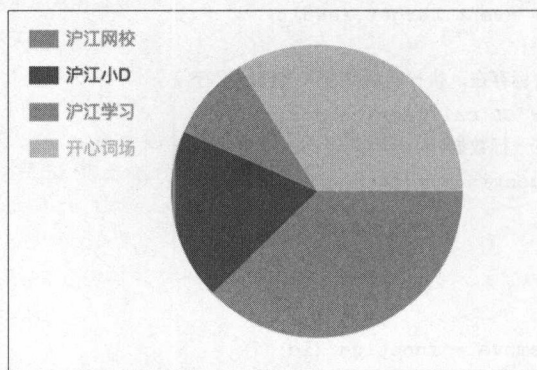


图 3.18 使用 Canvas 绘制一个简单的饼图

要实现上图中饼图效果, 首先在 HTML 引入 Canvas 标签, 代码如下:

```
<canvas class="pie-chart" width="850" height="500"></canvas>
```

紧接着引入 JavaScript 文件, 创建 PieChart 类, 并在其构造函数中获取 Canvas 的 Context 环境, 代码如下:

```
01 let PieChart = function(selector, options) {
02     let canvas = "string" === typeof selector ? document.querySelector(selector) : null;
03     if(canvas === null) return false;
04     let defaultOptions = {
05         radius: 200, // 饼图半径
06         legendParms: { // 图例参数
07             font: "24px Arial", // 图例字体属性
08             x: 30, // 图例 x 轴坐标
09             y: 30, // 图例 y 轴坐标
```

```

10         margin: 50,                                // 图例间距
11         width: 40,                                   // 图例宽度
12         height: 24                                   // 图例高度
13     }
14 }
15 this.context = canvas.getContext("2d");              // 获取 context 环境
16 this.width = canvas.getAttribute("width") || 300;
17 this.height = canvas.getAttribute("height") || 300;
18 this.options = Object.assign(defaultOptions, options); // 合并参数
19 };

```

添加 **PieChart** 类原型方法 **load** 用于载入饼图所使用的数据，并计算饼图的数据总量，用于之后渲染饼图时分配每个数据所对应的扇形比例，代码如下：

```

01 PieChart.prototype.load = function(data) {
02     data.forEach(item => this.count ? this.count += item.value : this.count =
    item.value);
03     this.data = data;
04     return this;                                     // 实现链式调用
05 };

```

添加 **PieChart** 类原型方法 **render** 用于对饼图进行渲染，**_generateLegend** 内部函数用于创建饼图对应的图例，当存在 **legend** 参数时调用 **_generateLegend** 生成饼图图例，代码如下：

```

01 PieChart.prototype.render = function() {
02     let _generateLegend = (item, index) => {          // 绘制图例方法
03         this.context.fillRect(                        // 绘制图例图标
04             this.options.legendParms.x,
05             this.options.legendParms.y + index * this.options.legendParms.margin,
06             this.options.legendParms.width,
07             this.options.legendParms.height
08         );
09         this.context.font = this.options.legendParms.font;
10         this.context.fillText(                         // 绘制图例文字
11             item.title,
12             this.options.legendParms.y + this.options.legendParms.margin,
13             (index + 1) * this.options.legendParms.margin
14         );
15     };
16     let temparc = 0;
17     this.data.forEach((item, index) => {              // 遍历绘制饼图扇形区域
18         item.color = `#${('00000'+(Math.random()*0x1000000<<0).toString(16)).substr(-6)}`;

```

```

19     this.context.beginPath();
20     this.context.moveTo(this.width / 2, this.height / 2);
21     let startarc = temparc, endarc = startarc + (item.value / this.count) * Math.PI * 2;
22     this.context.arc(                                // 饼图弧形区域
23         this.width / 2,                                // 圆中心点 x 坐标
24         this.height / 2,                                // 圆中心点 y 坐标
25         this.options.radius,                            // 饼图半径
26         startarc,                                        // 开始角度
27         endarc,                                        // 结束角度
28         false                                          // 逆时针
29     );
30     this.context.closePath();
31     this.context.fillStyle = item.color;
32     this.context.fill();                                // 填充路径
33     temparc = endarc;
34     if (this.options.legend) {                          // 是否需要绘制图例
35         generateLegend(item, index);
36     }
37 });
38 return this;
39 };

```

最后，引入需要绘制的数据创建饼图对象即可完成饼图绘制，代码如下：

```

01 const data = [
02     {title: "沪江网校", value: 1024},
03     {title: "沪江小 D", value: 512},
04     {title: "沪江学习", value: 256},
05     {title: "开心词场", value: 920}
06 ];
07 let pie = new PieChart(".pie-chart", {legend: true});
08 pie.load(data).render();

```

注意：Canvas 绘制的图像在一些设备 devicePixelRatio（设备像素比）不为 1 的 Retina 屏幕中显示会出现模糊，要处理这个问题，一种简单的方式是为 Canvas 元素设置 CSS transform 属性来缩放原有的 Canvas 元素。

3.4.2 使用 SVG 实现奥运五环

SVG，即可缩放矢量图形（Scalable Vector Graphics），基于 XML，是用于描述二维矢量图形的一种图形格式。由于 SVG 可缩放的矢量图形特性，意味着在对 SVG 图形做任意尺寸变化时，

不会出现模糊和失真。本节将利用 SVG 的功能绘制一个简单的奥运五环的效果,如图 3.19 所示。

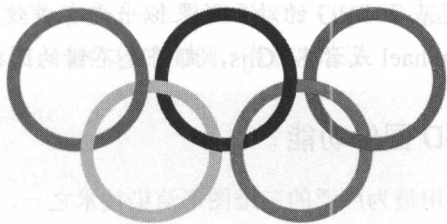


图 3.19 使用 SVG 实现奥运五环

从效果不难看出,要实现奥运五环的排列,可以利用 SVG 内置的 Circle 标签。Circle 标签元素可以基于中心点和半径绘制一个圆环,整个实例需要引入五个 Circle 标签元素,代码如下:

```
<svg width="450" height="300" xmlns="http://www.w3.org/2000/svg">
  <g stroke-width="10" fill="none">
    <circle stroke="#0085c7" r="50" cy="117" cx="105"/>
    <circle stroke="#000000" r="50" cy="117" cx="220"/>
    <circle stroke="#df0024" r="50" cy="117" cx="335"/>
    <circle stroke="#f4c300" r="50" cy="172" cx="162"/>
    <circle stroke="#009f3d" r="50" cy="172" cx="278"/>
  </g>
</svg>
```

G 标签是 SVG 的容器元素,可以将相关的元素组合在一起进行绘制。由于 SVG 根据元素的先后顺序绘制,上述代码只是简单排列了五环并以层叠的方式展示,并不能实现五环环环相扣的效果,要让五环环环相扣,一种简单粗暴的方式是进行“补刀”,利用 Path 标签或者 Line 标签绘制一个新的圆弧或者线段制造视觉差异。因此,选择将新增加的 Line 标签元素添加到实例中,代码如下:

```
<g stroke-width="10" fill="none">
  <line stroke="#000000" x1="270" y1="116" x2="268" y2="132"/>
  <line stroke="#0085c7" x1="156" y1="116" x2="153" y2="130"/>
  <line stroke="#df0024" x1="317" y1="163" x2="335" y2="168"/>
  <line stroke="#000000" x1="204" y1="165" x2="218" y2="168"/>
</g>
```

在经过“补刀”之后,一个环环相扣的奥运五环就展现在面前了。当然,也可以利用 Path 标签绘制圆弧来实现奥运五环,读者可以自行尝试使用,这里不再赘述。

注意: 由于 SVG 基于 XML 的特性, 在实际运用中显得并不那么友好, 可以选用一款 SVG 编辑器来完成图形的绘制。当然, 对于前端工程师来讲, 使用 JavaScript 绘制 SVG 以及实现基于 SVG 的动画效果似乎更有成效, 主流的 SVG JavaScript 类库, 如 Snap、Raphael 或者 SVG.js, 都将是不错的选择。

3.4.3 WebGL 带来了 3D 图像功能

OpenGL 是计算机领域使用最为广泛的三维图形渲染技术之一, 而 WebGL 的技术规范则是继承自 OpenGL 标准, 从某种意义上来说, WebGL 可以认为是 Web 版的 OpenGL。

在前面的章节中, 已经介绍过在 Canvas 中绘制 2D 图形的方法, 与之前使用 Canvas API 在 Canvas 2D Context 中绘制图形不同, WebGL 使用基于 OpenGL ES (OpenGL for Embedded Systems, OpenGL 的子集) 的 API 可以在 Canvas 中进行 3D 图形渲染。由于在 WebGL 中处理的是三维图形, 所以 WebGL 默认使用了右手坐标系的三维坐标系统, 由 Z 轴来表示深度, 如图 3.20 所示。

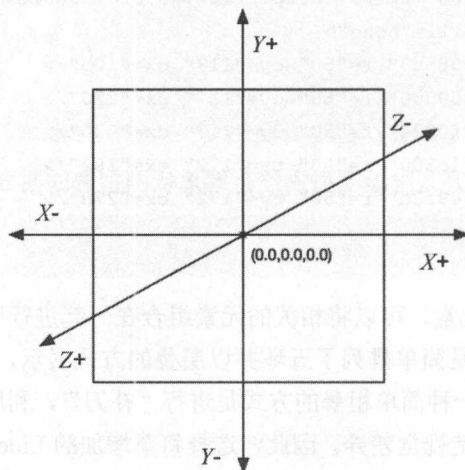


图 3.20 WebGL 坐标系

注意: 准确来讲, OpenGL ES 并不强制使用左手或右手坐标系, 而右手坐标系是大多数 GL (Graphics Library) 图形库由来已久的惯例。在 WebGL 中, 也有可能使用到左手坐标系, 如开启隐藏面消除时使用的裁剪坐标系。

WebGL 依赖于一种称之为着色器 (Shader) 的机制来绘制图形, 要使用 WebGL 绘制图形, 就必须使用着色器。着色器可以理解为提供给计算机执行图形渲染任务时使用的指令, 也就是告诉计算机如何使用特定的方法去绘制物体。在 WebGL 中有两种着色器, 一种称为顶点着色器

(Vertex Shader)，主要用来描述顶点（二维或三维空间中的端点或交点）特征，如位置、坐标等信息；另一种称为片段着色器（Fragment Shader），主要用来处理颜色、纹理等信息。在 WebGL 中编写着色器使用 GLSL ES 语言，GLSL ES 是在 GLSL（OpenGL Shading Language）基础上删减简化而来。下面看一个简单的顶点着色器的实例，代码如下：

```
attribute vec4 aVertexPosition;
void main() {
    gl_Position = aVertexPosition;
    gl_PointSize = 10.0;
}
```

上面的代码中，声明了 `vec4` 类型的顶点位置 `aVertexPosition`，关键词 `attribute` 是存储限定符，表示可以从 JavaScript 中接受 `attribute` 变量。着色器代码以字符串的形式传递给 JavaScript，并通过 JavaScript 创建、编译并完成着色器的链接工作。

由于 WebGL 的知识体系需要对图形学有更多深入的了解，这里就不再过多展开。为了更好地展示 WebGL 带来的 3D 图像功能，接下来将使用 Three.js 实现一个简单的正方体 3D 图形，效果如图所示 3.21 所示。

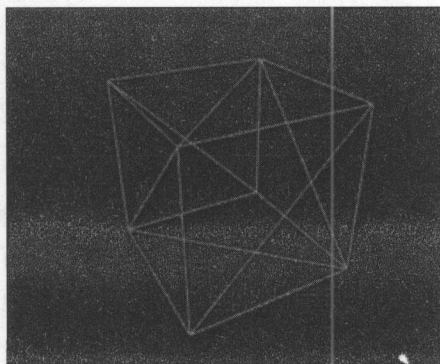


图 3.21 使用 Three.js 绘制简单的正方体

使用 Three.js 绘制上图中的正方体，首先需要使用 Three.js 创建 WebGL 的图形渲染器，代码如下：

```
var renderer = new THREE.WebGLRenderer({
    antialias:true // 设置抗锯齿效果
});
renderer.setSize(400, 300); // 设置渲染器尺寸
document.body.appendChild(renderer.domElement); // 将对应 canvas 插入文档
```

由于在 Canvas 中所看到的三维空间并非真实的三维空间,而是用数学算法将模拟的三维空间投射到二维视口的图像。在 WebGL 中,三维空间中的物体投影到二维空间分为正交投影和透视投影两种方式,正交投影就是不管物体和视点距离,都按照统一的大小进行绘制,而透视投影则是从视点开始越近的物体越大、远处的物体则绘制的较小。同样,Three.js 也能够设置这两种投影方式,分别对应 OrthographicCamera 与 PerspectiveCamera 两个方法,那么接下来使用透视投影的方式在 WebGL 中引入透视投影相机,代码如下:

```
var camera = new THREE.PerspectiveCamera( 60, 400 / 300, 1, 5000 );
camera.position.z = 500;
```

相机已经到位,接下来的事情就是创建场景,并在场景中加入本次拍摄的主角,一个带线框效果的正方体,代码如下:

```
var scene = new THREE.Scene(); // 新建场景
var cube = new THREE.Mesh( // 新建图形
    new THREE.BoxGeometry( 200, 200, 200 ), // 创建正方体
    new THREE.MeshBasicMaterial( { color: 0x6699cc, wireframe: true } ) // 设置材质
);
cube.rotation.x = 0.5; // x轴旋转
cube.rotation.y = 0.5; // y轴旋转
scene.add(cube);
```

最后将在已经准备好的场景中进行“拍摄”,完成实例中的正方体效果,代码如下:

```
renderer.render( scene, camera );
```

不难看出 Three.js 大大降低了 WebGL 编程的难度,也因此被誉为前端图形领域的 jQuery。Three.js 官方有大量的实例以及完备的文档,读者可以自行查阅,而整个 WebGL 所涵盖的知识领域相当庞杂,如果希望对 WebGL 有更加深入的了解,需要阅读其他专门介绍 WebGL 知识体系的书籍。

3.5 不一样的通信

3.5.1 PostMessages

在过去,跨源或者窗口之间的通信往往通过和服务器进行数据交互来实现,并且需要借助轮询或者 Comet 等技术来监听消息。HTML 5 提供了新型机制 PostMessages 实现安全的跨源通信。浏览器同源是指协议(如 HTTPS)、端口、域名相同,同时为了用户信息安全,浏览器在实现机制上使用同源策略进行限制,HTML 5 的新功能 PostMessages 在一定程度上兼顾了安全和灵活性,

实现跨源之间的消息传递。

PostMessages 语法如下：

```
otherWindow.postMessage(message, targetOrigin, [transfer]);
```

- otherWindow: 其他窗口的一个引用，比如 IFRAME 的 contentWindow 属性、执行 window.open 返回的窗口对象。
- message: 将要发送到其他窗口的数据。
- targetOrigin: 通过窗口的 origin 属性来指定哪些窗口能接收到消息事件，其值可以是字符 “*”（表示无限制）或者一个 URI。
- transfer: 是一串和 message 同时传递的 Transferable 对象。这些对象的所有权将被转移给消息的接收方，而发送一方将不再保有所有权。该参数可选。

通过一个实例向读者介绍 PostMessages 的使用，首先是数据发送功能，创建一个数据发送的文件（即数据源），可以使用 window.open 方法打开一个新窗口或者创建一个 IFRAME，然后在新窗口中发送消息。本例使用 IFRAME 做演示，代码如下：

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     <title>消息发送端</title>
06 </head>
07 <body>
08     <script type="text/javascript">
09         ;(function() {
10             // 模拟数据
11             var messages = [
12                 '今天天气不错',
13                 '明天的会议大家不要迟到',
14                 '今晚大家去吃一顿好的',
15                 '打车记得拿发票',
16                 '明天请假一天',
17                 '这本书干货很多，大家好好看'
18             ];
19             // 随机获取 message 信息，真实环境是从服务端获取数据
20             var getMessage = function() {
21                 var index = Math.floor(Math.random() * 10);
22                 // 如果数据不存在则返回 null
23                 return messages[index] || null;
```

```

24         };
25         var postMessageLoop = function() {
26             var randomTime = Math.floor(Math.random() * 10000);
27             setTimeout(function() {
28                 var message = getMessage();
29                 if(message !== null) {
30                     // 如果消息不为 null，则发送消息到父页面
31                     window.parent.postMessage(message, 'http://localhost:8080');
32                 }
33                 postMessageLoop();
34             }, randomTime);
35         };
36         postMessageLoop();
37     }());
38 </script>
39 </body>
40 </html>

```

接着是数据接收端，创建一个数据接收文件，用于将接收数据显示于页面，代码如下：

```

01 <!DOCTYPE html>
02 <head>
03     <title>消息接收端</title>
04     <style type="text/css">
05         #messageList {
06             border: 1px solid #333;
07             border-radius: 4px;
08             width: 200px;
09             min-height: 150px;
10         }
11         #postWindow { display: none; }      /* 设置 iframe 不可见 */
12     </style>
13 </head>
14 <body>
15     <h3>消息接收端</h3>
16     <ul id="messageList"></ul>
17     <iframe id="postWindow" src="post_page.html"></iframe>
18     <script type="text/javascript">
19         ;(function(W) {
20             var doc = W.document;
21             var msgList = doc.querySelector('#messageList');
22             var handler = function(msg) {      // 处理新的消息

```



```

23         var li = doc.createElement('li');
24         li.innerText = msg;
25         msgList.appendChild(li);           // 把消息显示在消息列表中
26     };
27     // 监听 postMessage 发送的消息
28     W.addEventListener('message', function(evt) {
29         if(evt.origin === 'http://localhost:8080') { // 判断消息的来源是否正确
30             handler(evt.data);           // 处理新的消息
31         }
32     }, false);
33     }(window));
34 </script>
35 </body>
36 </html>

```

将代码文件部署在本地 Web 服务器。如果电脑上已经装有 Python，可以在根目录通过以下命令快速启动一个本地服务，命令如下：

```
python -m SimpleHTTPServer 8080
```

通过命令在 8080 端口启动一个 HTTP 服务，启动之后可以通过访问地址 http://localhost:8080/receive_page.html 查看页面效果，如图 3.22 所示。

消息接收端

- 今天天气不错
- 明天的会议大家不要迟到
- 这本书干货很多，大家好好看
- 今天天气不错
- 这本书干货很多，大家好好看

图 3.22 接收消息端效果

每隔若干秒，消息接收页面就会接收到新的消息，并且在页面中进行展示。这个功能常常可以用在微博信息的同步或者协同办公系统的消息同步中。

3.5.2 XMLHttpRequest Level 2

XMLHttpRequest 为浏览器提供了在客户端和服务器之间传输数据的功能，即通常所说的 AJAX，用于页面无刷新更新数据，进而提升产品交互体验。

XMLHttpRequest Level 2 相较于老版本 XMLHttpRequest Level 1 做出了大幅的改进, 主要包括如下几点(代码中的 xhr 表示创建的 XMLHttpRequest 实例)。

- 设置 HTTP 请求的超时时间, 代码如下:

```
xhr.timeout = 3000;
xhr.ontimeout = function(event){
    alert('请求超时! ');
}
```

- 使用 FormData 对象管理表单数据, 代码如下:

```
var formData = new FormData();
formData.append('username', '张三');
formData.append('id', 123456);
xhr.send(formData);
```

- 用于上传文件, 代码如下:

```
// 假定 files 是一个"选择文件"的表单元素 (input[type="file"])
var formData = new FormData();
for (var i = 0; i < files.length; i++) {
    formData.append('files[]', files[i]);
}
```

- 跨域请求, 需要浏览器支持, 并且服务器进行对应设置。
- 获取服务器端的二进制数据, 代码如下:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png');
// 把 responseType 设为 blob, 表示服务器传回的是二进制对象
xhr.responseType = 'blob';
```

- 获得数据传输的进度信息, 代码如下:

```
// 下载
xhr.onprogress = updateProgress;
// 上传
xhr.upload.onprogress = updateProgress;
function updateProgress(event) {
    if (event.lengthComputable) {
        // event.total 是需要传输的总字节, event.loaded 是已经传输的字节
        var percentComplete = event.loaded / event.total;
    }
}
```

通过实例介绍 XMLHttpRequest Level 2 的使用，本实例分为客户端和服务端，服务端采用 Node.js 实现。

客户端部分代码如下所示：

```

01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     <title>客户端</title>
06 </head>
07 <body>
08     数据: <input /><button>获取</button> <!-- 数据获取显示 -->
09     <script type="text/javascript">
10         // 监听按钮单击事件
11         document.querySelector('button')
12         .addEventListener('click', function(e){
13             // 阻止按钮默认提交事件
14             e.preventDefault();
15             // 实例化 XMLHttpRequest 对象
16             var xhr = new XMLHttpRequest();
17             // 判断浏览器是否支持 level 2
18             if(typeof xhr.withCredentials === undefined) {
19                 console.log('浏览器不支持 html5 XMLHttpRequest Level 2 的跨域请求');
20             } else {
21                 // 监听 load 事件
22                 xhr.onload = function() {
23                     // 将文本转化为 json 数据
24                     var data = JSON.parse(xhr.responseText);
25                     // 显示返回数据
26                     document.querySelector('input').value = data.data;
27                 }
28                 // 监听错误事件
29                 xhr.onerror = function(e) {
30                     console.log(e);
31                 }
32                 // 请求地址和方法
33                 xhr.open('GET', 'http://localhost:4412', true);
34                 // 发送请求
35                 xhr.send();
36             }

```

```

37         });
38     </script>
39 </body>
40 </html>

```

服务端的代码如下所示:

```

01 // 引用 http 模块, 用于创建 web 服务器
02 var http = require('http');
03 // 创建新服务器
04 http.createServer(function(req, res){
05     // 设置可以跨域的域名
06     res.setHeader('Access-Control-Allow-Origin', 'http://localhost:8080');
07     // 服务器支持"GET POST"方法
08     res.setHeader('Access-Control-Allow-Methods', 'GET,POST');
09     // 设置接收数据编码格式为 utf-8
10     req.setEncoding('utf8');
11     // 返回测试数据
12     res.end(JSON.stringify({data: 'Hello World!'}));
13 }).listen(4412, function(){
14     console.log('listening on http://localhost:4412');
15 });

```

运行本实例, 需要将客户端页面部署在 Web 服务器上, 如 IIS、Apache、Nginx 等, 或者在根目录下使用 Python 的命令快速启动一个简易 Web 服务器, 命令如下:

```
python -m SimpleHTTPServer 8080 // 端口可以根据实际情况调整
```

服务端直接使用 Node.js 启动, 命令如下:

```
node server.js
```

Web 服务器和 Node.js 服务启动之后, 在 Chrome 浏览器中打开 `http://localhost:8080/client.html`, 如图 3.23 所示。单击“获取”按钮, 从服务端获取数据, 如图 3.24 所示。

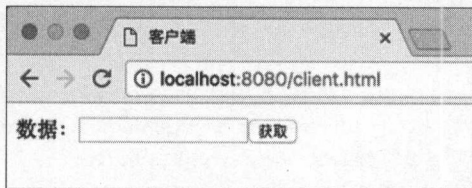


图 3.23 客户端效果

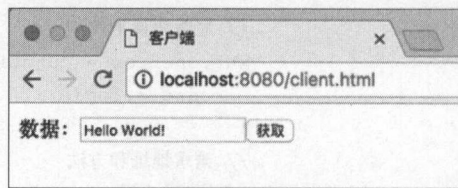


图 3.24 获取服务端数据

本实例主要演示了 XMLHttpRequest 的基本用法和 XMLHttpRequest Level 2 的跨域请求功能。

3.5.3 Server Sent Event

在通常情况下，客户端主动向服务端查询数据，并把数据显示到客户端界面。但在有些情况下，也需要服务端主动向客户端发送数据，并更新客户端信息，让用户始终能够看到最新信息。

比如说邮件应用，当新的邮件到达，服务端主动发送数据到客户端，提醒客户端有新的邮件，用户就能够及时处理信息。传统的做法是客户端向服务端发送轮询请求，一旦有新的数据，马上更新，这种做法消耗性能并且时效性差。HTML 5 中提供了 Server Sent Event 来处理这件事情。

Server Sent Event 技术有以下优点：

- 轻量，相对简单
- 单项传送数据（服务端向客户端传送）
- 基于 HTTP 协议
- 默认支持断线重连
- 自定义发送数据类型

Server Sent Event 通过 EventSource 对象接收服务器发送事件的通知，基本语法如下：

```
// 实例化 EventSource 对象
var source = new EventSource("demo_sse.php");
// 监听 message 事件，获取服务端的数据
source.onmessage = function(event) {
    document.getElementById("result").innerHTML += event.data + "<br>";
};
```

在平时的使用中还要判断浏览器是否支持 EventSource 对象，如果不支持需要降级处理，代码如下所示：

```
if(typeof(EventSource) !== "undefined") {
    // 支持服务器发送事件！
    // 一些代码.....
} else {
    // 抱歉！不支持服务器发送事件！
}
```

下面通过实例演示 Server Sent Event 的功能，先创建一个客户端文件 sse.html，用于初始化 Server Sent Event，并且监听 message、open、error 这三类事件处理业务逻辑，代码如下所示：

```
01 <!DOCTYPE html>
02 <head>
03     <title>Server-Sent Events Demo</title>
```

```
04 <script>
05 // 监听 load 事件, 页面 load 完成之后进行后续操作
06 window.addEventListener("load", function() {
07     // 缓存 DOM 对象
08     var status = document.getElementById("status");
09     var output = document.getElementById("output");
10     var source;
11     function connect() {
12         source = new EventSource("stream"); // 向服务端建立连接
13         // 监听 message 事件, 获取服务端发送的数据
14         source.addEventListener("message", function(event) {
15             output.textContent = event.data;
16         }, false);
17         // 监听 open 事件, 判断连接是否进行中
18         source.addEventListener("open", function(event) {
19             status.textContent = "连接打开了!";
20         }, false);
21         // 监听 error 事件, 处理连接错误的情况
22         source.addEventListener("error", function(event) {
23             if (event.target.readyState === EventSource.CLOSED) {
24                 source.close();
25                 status.textContent = "连接关闭了!";
26             } else {
27                 status.textContent = "连接关闭了!未知错误!";
28             }
29         }, false);
30     }
31     if (!window.EventSource) { // 判断浏览器是否支持 Server Sent Event
32         connect();
33     } else {
34         status.textContent = "对不起, 你的浏览器不支持 server-sent events";
35     }
36 }, false);
37 </script>
38 </head>
39 <body>
40     <span id="status">Connection closed!</span><br />
41     <span id="output"></span>
42 </body>
43 </html>
```

接着, 创建服务端文件用于主动发送信息, 本例采用 Node.js 作为服务器, 代码如下所示:


```

01 var http = require("http");           // 引入 http 模块, 创建 Web 服务器
02 var fs = require("fs");               // 引入 fs 模块, 操作文件
03 http.createServer(function (req, res) {
04     var index = "./sse.html";         // 默认页面
05     var fileName;                     // 文件名
06     var interval;                     // 定时器
07     if (req.url === "/" ) {           // 判断 url 是什么
08         fileName = index;
09     } else {
10         fileName = "." + req.url;
11     }
12     if (fileName === "./stream") { // 如果 Server Sent Event 则设置相应头信息
13         res.writeHead(200, {
14             "Content-Type": "text/event-stream",
15             "Cache-Control": "no-cache",
16             "Connection": "keep-alive"
17         });
18         res.write("retry: 10000\n");    // 过 10000 秒重试
19         res.write("data: " + (new Date()) + "\n\n"); // 首先发送一次时间信息
20         interval = setInterval(function() { // 每隔 1 秒发送一次时间信息
21             res.write("data: " + (new Date()) + "\n\n");
22         }, 1000);
23         // 监听 close 事件, 用于停止定时器
24         req.connection.addListener("close", function () {
25             clearInterval(interval);
26         }, false);
27     } else if (fileName === index) {
28         // 判断是否为页面请求, 并找到相应文件返回页面
29         fs.exists(fileName, function(exists) {
30             if (exists) {
31                 .readFile(fileName, function(error, content) {
32                     if (error) {
33                         res.writeHead(500); // 文件查找失败返回 500
34                         res.end();
35                     } else {
36                         // 监听 close 事件, 用于停止定时器
37                         res.writeHead(200, {"Content-Type": "text/html"});
38                         res.end(content, "utf-8");
39                     }
40                 });
41             } else {

```

```

42         res.writeHead(404);           // 文件不存在返回 404
43         res.end();
44     }
45     });
46     } else {
47         res.writeHead(404);           // 路径不存在返回 404
48         res.end();
49     }
50 }).listen(8080, "127.0.0.1");
51 console.log("Server running at http://127.0.0.1:8080/");

```

运行启动 Node.js 服务, 在根目录下输入如下命令:

```
node sse_server.js
```

然后在 Chrome 浏览器中打开 <http://localhost:8080/sse.html>, 如图 3.25 所示。

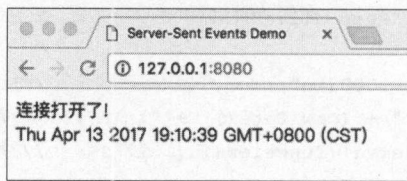


图 3.25 客户端接收数据效果

在 Chrome 浏览器中可以看到页面上的时间每隔 1 秒更新一次, 这是服务端主动发送数据的结果。

3.5.4 WebSocket

WebSocket 是 HTML 5 新增的协议, 基于 TCP 连接进行全双工通信。全双工是通信传输的一个术语, 表示允许数据在两个方向上同时传输。如果把 HTTP 协议比作发送电子邮件, 发出后必须等待对方回信。那么 WebSocket 就可以比作打电话, 服务器和客户端可以同时向对方发送数据。

WebSocket 对象提供了一组用于创建和管理 WebSocket 连接, 以及可以通过该连接发送和接收数据的 API。创建一个 WebSocket 实例对象, 代码如下:

```
var Socket = new WebSocket(url, [protocol] );
```

以上代码中的第一个参数指定连接的 URL。第二个参数 protocol 是可选的, 指定了可接受的子协议, 例如 SOAP 和 WAMP 协议等, 其他子协议可以参考 <http://www.iana.org/assignments/websocket/websocket.xml>。子协议是对 WebSocket 协议的一个补充, 用于标识服务端和客户端之间

如何进行协商。

在使用 WebSocket 的时候要先判断浏览器是否支持 WebSocket，代码如下所示：

```
// 判断浏览器是否支持 WebSocket
if(window.WebSocket != undefined) {
    var Socket = new WebSocket('ws://localhost:8080');
}
```

上述代码中 ws 协议是 WebSocket 新增的协议，此外，还有 wss 协议，表示加密的 WebSocket 协议。两者关系如同 HTTP 协议对应 HTTPS 协议。

WebSocket 提供了多种事件用于监听数据传输，实例代码如下：

```
// 创建 WebSocket 连接
var Socket = new WebSocket('ws://localhost:8080');
// 连接打开
Socket.addEventListener('open', function (event) {
    socket.send('Hello Server!');
});
// 监听数据的传送，有数据到达时会触发
Socket.addEventListener('message', function (event) {
    console.log('Message from server', event.data);
});
// 当错误发生时用于监听 error 事件的事件监听器
Socket.addEventListener('error', function (event) {
    console.log('Message from server', event.data);
});
// 用于监听连接关闭事件监听器
Socket.addEventListener('close', function (event) {
    console.log('Message from server', event.data);
});
```

WebSocket 可以替代 AJAX 技术，并且功能比 AJAX 更加强大。可以使用 WebSocket 开发即时聊天、互动游戏、股票信息等应用。

3.5.5 WebRTC

WebRTC 全称 Web Real-Time Communication，即 Web 实时通信，能够为浏览器和移动网页应用提供实时的语音或者视频通话功能。在前面的章节中，曾介绍过在网页中使用摄像头拍照，就是使用了 WebRTC 功能来实现。本节将简单地介绍 WebRTC 的几个主要的 API 和 WebRTC 的前景。

先来了解一下目前主要浏览器对 WebRTC 的支持情况，见表 3.6。

表 3.6 浏览器对 WebRTC 的支持

浏 览 器	版 本
Edge	15+
Firefox	22+
Chrome	23+
Oprea	18+
Android Browser	Android 5~6.x Chromium 56+
Chrome for Android	57+

提示：目前各浏览器对 WebRTC 的支持情况并不一致，部分浏览器实现的并非标准规范，为了兼容各浏览器的差异，WebRTC 组织提供了相应的 JavaScript 适配器，来保证使用 WebRTC 时 API 的一致性，下载地址为 <http://webrtc.github.io/adapter/adapter-latest.js>，也可以使用 NPM 工具，通过安装 webrtc-adapter 模块来获取。

WebRTC 包括以下几个主要的 JavaScript API。

- **MediaDevices:** 提供了查询和访问媒体输入设备的方法，其中包括前面 3.2.2 节介绍的捕获视频和音频的 `MediaDevices.getUserMedia` 方法。
- **RTCPeerConnection:** 提供建立点和点之间连接的方法，并维护和监听连接，建立连接的点和点之间可以传输视频流和音频流。
- **RTCDataChannel:** 可用于点和点之间双向传输任意数据的网络通道。由 `RTCPeerConnection` 建立，一个 `RTCPeerConnection` 可以建立理论最大值 65534 个数据通道。

注意：为了保证用户隐私的安全，所有的 WebRTC 组件都必须加密，WebRTC 推荐在安全源 HTTPS 协议下使用，这点同 Geolocation 功能相同，要确保使用安全协议。

下面是一个 WebRTC 使用的实例，从用户摄像头和麦克风获取音视频数据，并进行播放。单击“开始录制”按钮后开始录制媒体流，单击“停止录制”按钮会将视频中录制的数据进行播放。HTML 部分相关代码如下：

```

01 <body>
02     <video id="source" autoplay muted></video>           <!-- 显示摄像头的源视频 -->
03     <video id="recorded" autoplay loop></video>          <!-- 显示已录制的视频 -->
04     <div>
05         <button id="record" disabled>开始录制</button><!-- 播放按钮 -->
06     </div>
07     <!-- 提供各浏览器 WebRTC 相关 API 一致性的适配器 -->

```



```

08     <script src="https://webrtc.github.io/adapter/adapter-latest.js"></script>
09     <script src="main.js"></script>                                <!-- 应用相关的代码 -->
10 </body>

```

核心 JavaScript 文件 main.js 代码如下:

```

01 var mediaSource = new MediaSource();                                // 创建媒体数据源
02 // 添加媒体数据源打开时的监听
03 mediaSource.addEventListener('sourceopen', handleSourceOpen, false);
04 var mediaRecorder, recordedBlobs, sourceBuffer;                    // 声明变量
05 var sourceVideo = document.getElementById('source');              // 源视频
06 var recordedVideo = document.getElementById('recorded');          // 已录制视频
07 var recordButton = document.getElementById('record');             // 录制按钮
08 recordButton.onclick = toggleRecording;                            // 设置录制按钮单击动作
09 // 设置媒体约束, 接收声音和视频, 视频宽度为 320 像素
10 var constraints = { audio: true, video: { width: 320 } };
11 function handleSuccess(stream) {                                    // 成功获取用户媒体
12     recordButton.disabled = false;                                  // 设置录制按钮可用
13     window.stream = stream;
14     sourceVideo.srcObject = stream;                                // 将摄像头画面显示在 sourceVideo 上
15 }
16 function handleError(error) {                                       // 获取用户媒体异常
17     console.log('获取用户媒体错误: ', error);
18 }
19 // 获取用户媒体
20 navigator.mediaDevices.getUserMedia(constraints).then(handleSuccess).catch(handleError);
21 function handleSourceOpen(event) {                                    // 处理媒体源打开
22     sourceBuffer = mediaSource.addSourceBuffer('video/webm; codecs="vp8"');
23 }
24 function handleDataAvailable(event) {                                // 处理数据可用
25     if (event.data && event.data.size > 0) {
26         recordedBlobs.push(event.data);                            // 将数据追加到录制记录中
27     }
28 }
29 function toggleRecording(){                                          // 切换录制
30     if (recordButton.textContent === '开始录制') {
31         startRecording();                                            // 开始录制
32     } else {
33         stopRecording();                                             // 停止录制
34         recordButton.textContent = '开始录制';
35     }
36 }

```



```

37 function startRecording() { // 开始录制
38     recordedBlobs = []; // 数据记录初始化
39     var mimeTypes = [
40         'video/webm;codecs=vp9',
41         'video/webm;codecs=vp8',
42         'video/webm'
43     ];
44     // 查找支持的视频格式
45     var mimeType = mimeTypes.find(type=>MediaRecorder.isTypeSupported(type)) || '';
46     try {
47         // 创建媒体录制器
48         mediaRecorder = new MediaRecorder(window.stream, { mimeType });
49     } catch (e) {
50         alert('创建媒体录制器异常 ' + options.mimeType);
51         return;
52     }
53     recordButton.textContent = '停止录制';
54     mediaRecorder.ondataavailable = handleDataAvailable;
55     mediaRecorder.start(10);
56 }
57 function stopRecording() { // 停止录制
58     mediaRecorder.stop();
59     var buf = new Blob(recordedBlobs, { type: 'video/webm' });
60     // 设置已录制视频的源为录制好的视频
61     recordedVideo.src = window.URL.createObjectURL(buf);
62 }

```

运行效果如图 3.26 所示。

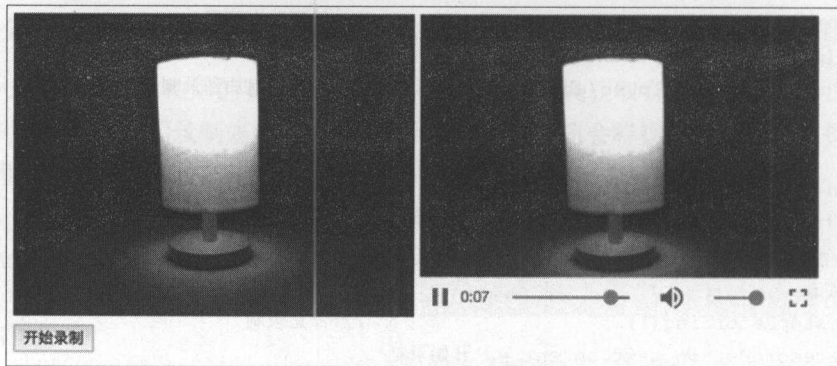


图 3.26 运行效果

提示：运行实例需要 Chrome 浏览器开启 Experimental Web Platform features 选项，可以从 `chrome://flags` 打开。

WebRTC 现今已然成为 Web 端最为重要的多媒体通信解决方案，可以不依赖浏览器插件（比如 Flash）实现基于浏览器建立音视频和数据的传输，为 Web 开发者提供了丰富多彩的实时多媒体功能。实时通信的背后是一系列的复杂的技术，包含音视频采集、编解码、网络传输等，但是有了 WebRTC 后，开发者只需要关注几个简单的 JavaScript API 即可。WebRTC 目前虽未普及到所有的浏览器中，但是已经被应用在很多项目和产品上了，相信这个颠覆性技术在未来会越来越重要。

3.6 其他常用特性

3.6.1 History API 与单页应用

单页应用（Single Page Application，简称 SPA）是指 Web 应用可以无刷新在不同的页面间切换，并且页面访问记录会被浏览器保存，从而支持浏览器的前进、后退和刷新等操作。

HTML 5 在 History 对象上新增了 `pushState` 和 `replaceState` API，配合在 `window` 对象上新增的 `popState` 事件使用，可以实现单页应用功能。

本节实例将使用 History API 实现一个单页应用，页面访问路径为 `/green`，效果如图 3.27 所示。当使用路径 `/green` 来访问页面时，页面展示效果为“green”菜单项被选中，并且右边内容区域显示文字“this is a green page”。

当单击“blue”菜单项时，浏览器路径被修改为 `/blue`，页面展示效果为“blue”菜单项被选中，并且右边内容区域显示文字“this is a blue page”，如图 3.28 所示。

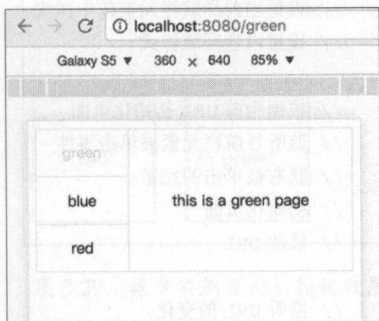


图 3.27 访问路径为 `/green` 的效果

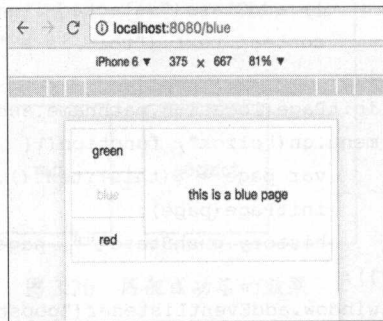


图 3.28 访问路径切换为 `/blue` 的效果

此时, 无论使用浏览器的回退还是刷新功能, 页面访问路径都能正常切换, 并且与页面展示效果保持一致。

注意: 单击不同菜单选项进行页面切换时, 浏览器不会重新加载页面。

下面介绍实例中 HTML 部分核心代码, 如下:

```
<div class="wrapper">                                <!-- 最外层包裹元素 -->
  <ul class="navigator">                             <!-- 左侧导航栏 -->
    <li class="nav-item">                             <!-- 导航栏菜单项 -->
      green
    </li>
    <li class="nav-item"                             <!-- 导航栏菜单项 -->
      blue
    </li>
    <li class="nav-item">                             <!-- 导航栏菜单项 -->
      red
    </li>
  </ul>
  <div class="content">                             <!-- 右侧内容区域 -->
  </div>
</div>
```

JavaScript 核心代码如下:

```
01 var menu = $("ul.navigator > li");                // 获取左侧导航栏元素
02 var content = $("div.content");                    // 右侧内容区域
03 function initPage(page) {                          // 根据当前 URL, 初始化页面
04   menu.removeClass("selected-item");                // 清除导航栏元素选中样式
05   menu.filter(function(){                           // 找到当前导航栏元素
06     return $(this).text().toLowerCase().trim() === page;
07   }).addClass("selected-item");                     // 设置当前导航栏元素选中样式
08   content.text ("this is a " + page + " page");     // 设置内容区域样式
09 }
10 initPage(location.pathname.substring(1));           // 根据当前 URL 初始化页面
11 menu.on("click", function(){                        // 监听导航栏元素被单击事件
12   var page = $(this).text().toLowerCase().trim(); // 获取被单击的元素
13   initPage(page);                                   // 初始化页面
14   history.pushState("", page, page);                // 修改 URL
15 });
16 window.addEventListener("popstate", function(e) {  // 监听 URL 的变化
17   initPage(location.pathname.substring(1));         // 根据当前 URL 初始化页面
18 });
```

注意：通常页面切换的时候，需要从服务端获取相关数据来渲染新页面，这里为了简化例子突出主要功能，并没有从服务器端拉取数据。

代码第 03~09 行定义了页面初始化的逻辑。代码第 10 行在页面加载时初始化页面。代码第 11~15 行响应导航栏的页面切换。代码第 16~18 行监听浏览器前进、后退事件。

CSS 代码和服务端 Node.js 代码与本节主题关系较弱，不在此展开讨论，本节完整代码见附带源码。

3.6.2 Drag 和 Drop 介绍

在没有提供 Drag 和 Drop 功能之前，开发者需要通过元素的 mousedown、mousemove、mouseup 等事件来实现拖放和拖曳效果。HTML 5 新增的 Drag 和 Drop 功能不仅另外提供了一套规范的事件格式，而且还支持桌面文件到浏览器的拖放，大大简化了开发复杂度。

本节实例将介绍如何在浏览器中实现拖曳效果，页面初始效果如图 3.29 所示。

页面中的圆形元素允许可拖动，将圆形元素拖动到正方形元素之上，并且释放元素，如果正方形元素内容为“green”，则两者匹配成功，圆形元素消失不见，正方形元素内容更改为“correct”。匹配成功后的效果如图 3.30 所示。

注意：由于 HTML 5 Drag 和 Drop API 的浏览器兼容性问题，建议读者在 Chrome 浏览器里运行。

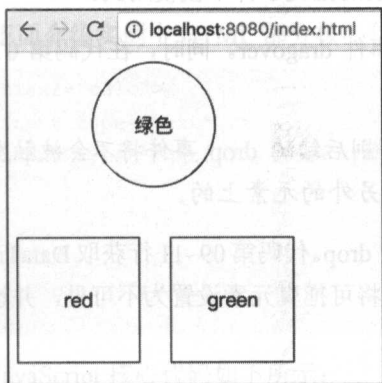


图 3.29 拖曳实例页面初始化效果

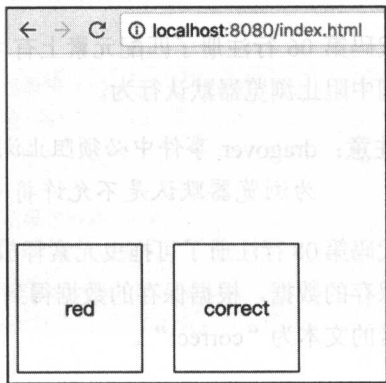


图 3.30 匹配成功后的效果

下面介绍实例中 HTML 部分核心代码，如下：


```

<div class="draggable">                                <!-- 包裹可拖曳元素 -->
  <div draggable="true">绿色</div>                    <!-- 圆形可拖曳元素 -->
</div>
<div class="text">                                      <!-- 包裹匹配元素 -->
  <div>red</div>                                       <!-- 正方形匹配元素 -->
  <div>green</div>                                     <!-- 正方形匹配元素 -->
</div>

```

实例中的圆形元素需要设置属性 `draggable` 为 `true`，表示元素可拖动。

JavaScript 核心代码如下：

```

01 var draggableColor = $(".draggable > div");          // 获取可拖曳元素
02 var redText = $(".text > div:nth-child(2)");         // 获取红色匹配元素
03 draggableColor.on("dragstart", function(event) {      // 注册拖曳开始事件
04     event.dataTransfer.setData("ele", ".draggable > div"); // 保存数据在后续事件中访问
05 })
06 redText.on("dragover", function(event) {              // 注册可拖曳元素经过事件
07     event.preventDefault();                           // 阻止浏览器默认行为
08 }).on("drop", function(event) {                       // 注册可拖曳元素释放事件
09     var dragTarget = event.dataTransfer.getData("ele"); // 获取保存的数据
10     $(dragTarget).css("visibility", "hidden");         // 隐藏可拖曳元素
11     $(this).text("correct");                           // 修改匹配元素的文本
12 })

```

代码第 03 行注册了可拖曳元素的开始拖曳事件 `dragstart`。同时，在代码第 04 行监听回调函数中，使用 `DataTransfer` 对象保存数据，这些数据将在后续的拖曳事件中被使用到。

代码第 06 行注册了匹配元素上有可拖曳元素经过的事件 `dragover`。同时，在代码第 07 行监听回调中阻止浏览器默认行为。

注意：`dragover` 事件中必须阻止浏览器默认行为，否则后续的 `drop` 事件将不会被触发。因为浏览器默认是不允许将可拖曳元素放置到另外的元素上的。

代码第 08 行注册了可拖曳元素释放到匹配元素的事件 `drop`。代码第 09~11 行获取 `DataTransfer` 对象保存的数据，根据保存的数据得到可拖曳元素，然后将可拖曳元素设置为不可见，并设置匹配元素的文本为“correct”。

注意：代码只对能正确匹配的元素进行了处理，不能正确匹配的元素因为没有任何特殊处理，所以会表现为默认的拖曳行为。CSS 代码跟本节主题关系不大，就不在此展开讨论，本节完整代码见附带源码。

3.6.3 利用 Web Workers 加速应用计算

JavaScript 是单线程运行，如果某个操作非常耗时，页面将会处于“假死”状态。Web Workers 赋予了 JavaScript 多线程运行的能力，可以将耗时操作放在 Web Workers 线程里运行，防止页面出现假死。

本节的例子是根据用户输入的值，计算对应的位置在斐波那契数列中的值。页面初始效果如图 3.31 所示。

用户在文本框内输入数值，单击“计算”按钮，计算值将显示在“计算结果”文本下方，如图 3.32 所示。

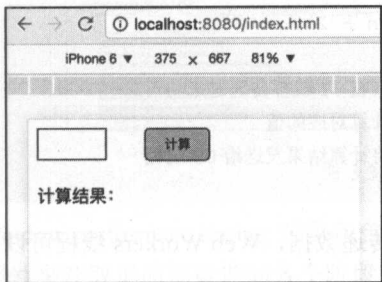


图 3.31 页面初始效果

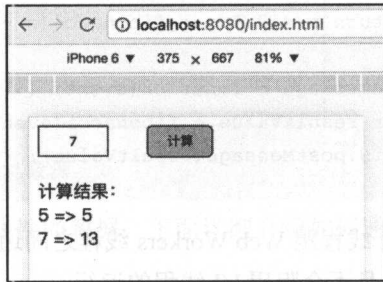


图 3.32 显示计算结果

如果输入较大的数值，比如 50，计算结果需要比较长的一段时间。但是因为计算的操作由 Web Workers 线程实现，不会影响 UI 线程，页面不会假死，还可以继续响应用户的其他操作。

下面介绍实例中 HTML 部分核心代码，如下所示：

```
<div class="calc">                                <!-- 输入包裹块 -->
  <input type="text" />                            <!-- 输入值 -->
  <input type="button" value="计算" /><!-- 计算按钮 -->
</div>
<div>                                              <!-- 计算结果包裹块 -->
  计算结果:
  <div class="result"></div>                        <!-- 计算结果 -->
</div>
```

JavaScript 核心代码如下所示：

```
var input = $("input[type='text']");              // 获取输入框
var cal = $("input[type='button']");              // 获取计算按钮
var result = $(".result");                        // 获取结果包裹块
```

```

cal.on("click", function(){ // 响应计算按钮单击事件
    var initValue = input.val(); // 获取输入值
    var w = new Worker("./worker.js"); // 创建 Web Workers 线程
    w.postMessage(initValue); // 给 Web Workers 线程发送消息
    w.onmessage = function(event) { // 接收 Web Workers 线程传回的消息
        // 设置计算结果
        result.html(result.html() + initValue + " => " + event.data + "<br/>");
    }
})

```

创建的 Web Workers 线程的入口文件为 worker.js，核心代码如下所示：

```

function fibonacci(n) { // 计算斐波那契数列
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
}
this.onmessage = function(event) { // 接收 UI 线程传递过来的消息
    var resultValue = fibonacci(event.data); // 计算对应的值
    this.postMessage(resultValue); // 将计算结果发送给 UI 线程
}

```

UI 线程跟 Web Workers 线程之间通过消息机制互相传递数据，Web Workers 线程可以启动多个，并且不会阻碍 UI 线程的运行。

注意：因为是多线程运行，计算结果的顺序取决于开始计算的时间和计算出结果需要的时间之和，与开始计算的顺序没有必然联系。

3.6.4 利用 Performance API 分析网站性能

网站的性能会直接影响到用户体验，是开发者应该重点关注的指标。早期搜集网站性能需要在页面里插入相应的脚本，监听页面不同时期的事件，比如 DOMContentLoaded 事件。这种方式的缺点是侵入性强，而且能够收集到的信息比较少，比如无法搜集 DNS 解析的时间。

HTML 5 里面新提供了可以获取页面加载详细性能指标的 Web Performance API，通过 window.performance 对象暴露给开发者。本节将会详细介绍如何通过 Web Performance API 分析网站性能。

注意：Web Performance API 是非侵入性的，也就是开发者不需要插入任何脚本，就可以直接使用 window.performance 对象获取页面性能数据。

访问沪江网校首页 <http://class.hujiang.com/>，打开调试器，在控制台输入如下代码：

```
window.performance.timing
```

运行结果如图 3.33 所示。

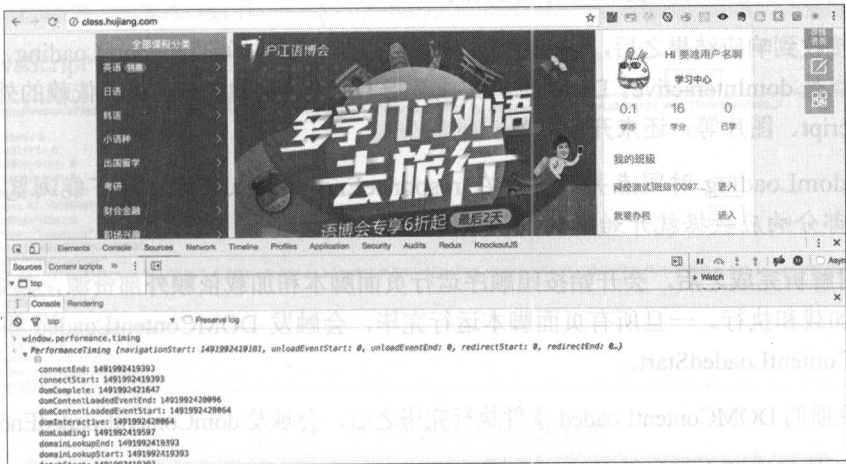


图 3.33 沪江网校首页性能数据

`window.performance.timing` 对象包含了完整的网页加载性能数据，下面详细介绍如何通过此对象各属性来获取页面加载期间各个阶段的性能。

页面加载的第一个时间点是 `navigationStart`，表示上一个页面的 `unload` 事件触发，接下来的时间点是 `fetchStart`，表示开始获取当前页面内容。`fetchStart` 时间点和 `navigationStart` 时间点之间的时间差是浏览器内核为加载新页面做的一些准备工作耗时。

获取页面内容的第一步是查询是否有跟页面相关的资源缓存，查询完毕之后，会触发开始 DNS 解析的时间点 `domainLookupStart`。`domainLookupStart` 时间点和 `fetchStart` 时间点之间的时间差是查询缓存所消耗的时间。

DNS 解析结束的时间点是 `domainLookupEnd`。`domainLookupEnd` 时间点和 `domainLookupStart` 时间点之间的时间差是 DNS 解析消耗的时间。

DNS 解析技术之后会开始建立 TCP 连接，TCP 连接开始和结束的时间点分别是 `connectStart` 和 `connectEnd`。`connectStart` 时间点紧接着 `domainLookupEnd` 时间点，`connectEnd` 时间点和 `connectStart` 时间点之间的时间差是建立 TCP 消耗的时间。

TCP 连接建立之后，开始发送请求内容至服务器端，这个时间点是 `requestStart`。服务器端接收到完整请求并处理完毕后，会将响应结果返回给客户端，开始发送响应结果的时间点为 `responseStart`。浏览器收到完整的响应结果之后，会触发 `responseEnd` 时间点。

注意: 没有服务器端开始收到请求和收到完整请求的时间点, 因为统计是在浏览器端进行, 浏览器无法获取该时间数据。

浏览器接收到响应结果之后, 会开始 DOM 树解析, 这个时间点是 `domLoading`, DOM 解析完成的时间点是 `domInteractive`。DOM 解析完成是指 DOM 树构建完成, 页面依赖的外部资源, 如 CSS、JavaScript、图片等, 还未开始加载。

注意: `domLoading` 时间点并不一定在 `responseEnd` 时间点之后, 有可能浏览器只接收了部分响应数据就开始解析 DOM 树。

DOM 树解析完成之后, 会开始按照顺序运行页面脚本和加载依赖外部资源, 其中 JavaScript 资源会同步加载和执行。一旦所有页面脚本运行完毕, 会触发 `DOMContentLoaded` 事件, 这个时间点是 `domContentLoadedStart`。

开发者注册的 `DOMContentLoaded` 事件执行完毕之后, 会触发 `domContentLoadedEnd` 时间点。

当依赖的外部资源全部加载并解析完成之后, 会触发 `domComplete` 时间点, 同时会触发暴露给开发者的 `load` 事件。`loadEventStart` 时间点表示 `load` 事件开始触发, `loadEventEnd` 时间点表示所有开发者注册在 `load` 事件上的脚本执行完毕。

至此, 整个页面加载的生命周期以及对应的性能分析方式已经介绍完毕。关于页面加载的生命周期, 可以参考 W3C 官网 <https://www.w3.org/TR/navigation-timing/#processing-model#processing-model>, 来自 W3C 官网的页面加载生命周期如图 3.34 所示。

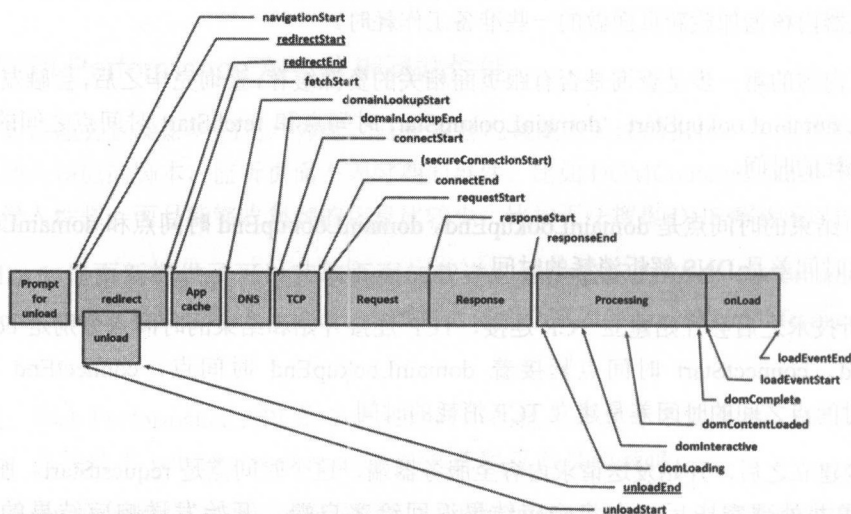
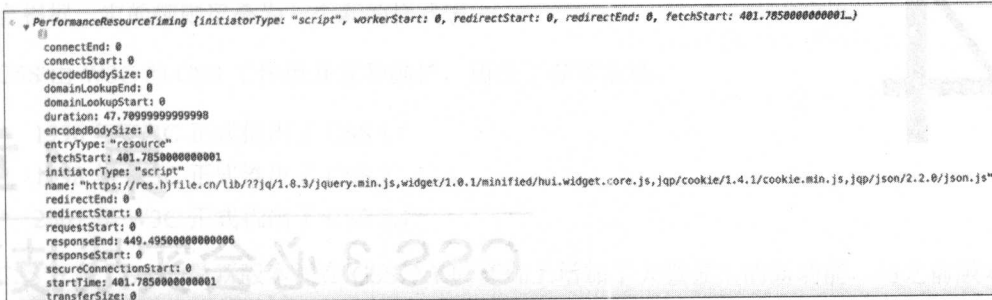


图 3.34 页面加载生命周期

除了页面本身的加载性能，还可以获取到所有依赖资源的加载性能，获取代码如下：

```
window.performance.getEntries()
```

某个 JavaScript 资源的性能数据如图 3.35 所示。



```
PerformanceResourceTiming {initiatorType: "script", workerStart: 0, redirectStart: 0, redirectEnd: 0, fetchStart: 401.7850000000001}
  connectEnd: 0
  connectStart: 0
  decodedBodySize: 0
  domainLookupEnd: 0
  domainLookupStart: 0
  duration: 47.789999999999998
  encodedBodySize: 0
  entryType: "resource"
  fetchStart: 401.7850000000001
  initiatorType: "script"
  name: "https://res.hjfile.cn/lib/??jq/1.8.3/jquery.min.js,widget/1.0.1/minified/hui.widget.core.js,jqp/cookie/1.4.1/cookie.min.js,jqp/json/2.2.0/json.js"
  redirectEnd: 0
  redirectStart: 0
  requestStart: 0
  responseEnd: 449.49500000000006
  responseStart: 0
  secureConnectionStart: 0
  startTime: 401.7850000000001
  transferSize: 0
```

图 3.35 单个资源的加载性能数据

3.7 本章小结

从 HTML 5 第一份正式草案 2008 年 1 月发布至今，已经经过了 9 年时光。在这 9 年中，HTML 5 规范仍在持续发展并已经被目前大多数主流的浏览器所支持。HTML 5 从语义、设备兼容特性、本地存储、图形特效、网页通信等各个方面，为 HTML 带来了全方位的改善。掌握 HTML 5 是每一名前端开发者必备的技能。

4

第 4 章

CSS 3 必会实战技巧

CSS 3 作为 CSS 技术的升级版本，经历多年的发展，已经在移动端被广泛使用，比如圆角、透明度、阴影、动画、响应式等功能。在使用上，做到了良好的优雅降级和渐进增强，成为移动 Web 开发不可或缺的利器，也是现今 Web 前端开发必须掌握的技能。

本章将从选择器、响应式开发、动效、预编译框架等方面，介绍日常开发实战中 CSS 3 的使用技巧，帮助读者掌握新特性，提升开发效率和质量。

4.1 认识 CSS 3

本节首先从 CSS 的发展历史讲起，介绍 CSS 3 概念及其发展现状。通过这部分的学习了解到 CSS 3 的新特性，接着讲解如何检测这些 CSS 3 新特性在浏览器中的兼容性。

4.1.1 什么是 CSS 3

CSS 全称 Cascading Style Sheets, 中文意思是层叠样式表, 是一种用来布局 and 美化网页的样式表语言。之所以称之为层叠样式表, 是因为当同一个 HTML 元素被多个样式表定义时, 所有的样式表会根据一定的规则层叠为一个新的样式表。

CSS 由 W3C 的 CSS 工作组开发和维护, 历经了多年发展。

- 1996 年 W3C 正式推出了 CSS 1
- 1998 年 W3C 正式推出了 CSS 2
- 2007 年 W3C 正式推出了 CSS 2.1

CSS 3 是 CSS 的最新版本, 在 CSS 2.1 的基础上增加了大量强大的新功能。与之前版本 CSS 不同的是, CSS 3 被划分为多个模块, 每个模块都有独立规范, 每个模块都有各自独立的创作者和发布时间表, 并且模块之间互不依赖。因为每个模块都被独立地标准化, 从形式上来说, 已经不存在 CSS 3 自身标准。

下面来看一下 CSS 3 包含哪些实用的新特性。

1. 选择器

CSS 选择器允许开发者选中特定的 HTML 元素, 而 CSS 3 新增的选择器可以减少多余的 Class、ID 或 JavaScript 的使用。表 4.1 列举了各大主流浏览器支持度较好并且开发中常用的 CSS 3 选择器。

表 4.1 常用 CSS 3 选择器

选 择 器	实 例	实例说明
[attribute^=value]	a[src^=http]	选择每一个src属性的值以“http”开头的元素
[attribute\$=value]	a[src\$=http]	选择每一个src属性的值以“http”结尾的元素
[attribute*=value]	a[src*=http]	选择每一个src属性的值包含字符串“http”的元素
:first-of-type	p:first-of-type	选择每个p元素是其父级的第一个p元素
:last-of-type	p:last-of-type	选择每个p元素是其父级的最后一个p元素
:only-of-type	p:only-of-type	选择每个p元素是其父级的唯一p元素
:only-child	:only-child	选择每个p元素是其父级的唯一元素
:nth-child(n)	p:nth-child(2)	选择每个p元素是其父级的第二个子元素

2. 盒模型

CSS 3 提供了 box-sizing 属性来改变默认的 CSS 盒模型对元素宽高的计算方式。默认值 content-box 是标准盒模型, width 和 height 只包括内容的宽和高。border-box 类型是 Internet Explorer 怪异模式 (Quirks Mode) 使用的盒模型, width 与 height 包括内边距与边框, 不包括外边距。

3. 个性化字体

@font-face 规则的引入, 允许开发者为其网页指定在线字体。这就突破了以往只能使用操作系统默认字体的限制, 使得网页能够在不使用图片的情况下, 运用更多个性化和美观的字体功能。@font-face 另外一个常见的用途是 IconFont, 把页面上一些简单的图标制作成字体, 获得字体自带的功能属性。沪江网校 IconFont 如图 4.1 所示。



图 4.1 沪江网校 IconFont

4. 自适应布局

CSS 3 提供了 calc 函数在渲染时动态计算属性值, 常被运用在自适应布局当中。假设需要实现一种布局, 父 container 包含子 header 和子 content。父 container 的高度在渲染时被确定(假设跟浏览器高度有关), 子 header 的高度永远是 50 像素, 其余部分为子 content 的高度。

此时运用 calc 函数来帮助布局会非常方便, 而且在语义上简单明了, 页面代码如下:

```
<html>
<style>
.container{                                     /* container 样式 */
    width: 200px;
    height: 200px;                             /* 值发生变化时, 整个布局保持不变 */
    background: green;
}
.header{ background: red; height: 50px; }      /* header 样式 */
.content{ background: blue; height: calc(100% - 50px); } /* container 样式 */
</style>
```

```
<div class="container">                                /* container 容器 */
  <div class="header"></div>                            /* header 容器 */
  <div class="content"></div>                          /* content 容器 */
</div>
</html>
```

实例运行效果如图 4.2 所示。

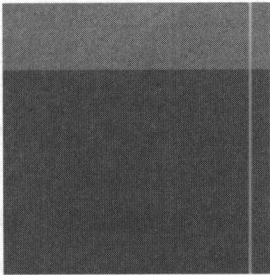


图 4.2 自适应布局实例效果

5. 其他常用 CSS 3 新特性

CSS 3 还提供了很多其他激动人心的新特性，例如：

- 圆角边框、字体阴影这种以前只能通过图片实现的效果。
- 响应式布局 Media Queries、弹性布局 Flexbox、多列布局 Multi-column Layout。
- 媲美原生应用的过渡与动画效果。

4.1.2 移动 Web 的 CSS 3 现状

上节提到 W3C 将 CSS 3 划分成了多种模块，每个模块都有各自的发布时间表。如果要知道 CSS 3 各模块在 W3C 的进度，可以访问网站 <https://www.w3.org/Style/CSS/current-work> 来了解详细情况。如图 4.3 所示，从这里可以看到每个模块当前所处进度、后续进度安排等。

Completed	Current	Upcoming	Notes	①	Title	Current	Notes
CSS Snapshot 2010	NOTE			①	Predefined Counter Styles	WD	I18N WG
CSS Snapshot 2007	NOTE			①	CSS Techniques for 'Web Content Accessibility Guidelines 1.0	NOTE	WCAG WG
CSS Color Level 3	REC	REC	See Errata	①	Associating Style Sheets with XML documents 1.0 (Second Edition)	REC	XML Core WG
CSS Namespaces	REC	REC		①	The 'view-mode' Media Feature	REC	Web Applications WG
Selectors Level 3	REC	REC		①	Selectors API Level 1	REC	Web Applications WG
CSS Level 2 Revision 1	REC	REC	See Errata	①	Selectors API Level 2	NOTE	Web Applications
CSS Level 1	REC		Unmaintained, see Snapshot	①			
CSS Print Profile	NOTE			①			
Media Queries	REC	REC		①			
CSS Style Attributes	REC	REC		①			

图 4.3 CSS 3 在 W3C 的现状

图 4.3 中的进度状态码简写、全称和中文解释, 如表 4.2 所示。

表 4.2 状态码、状态码全称与状态描述的对应关系

状态码	状态码全称	状态描述
WD	Working Draft	标准已经起草, 发布出来收集各方的意见
LC	Last Call	标准的意见收集进入最后阶段, 即将进入下一个状态
CR	Candidate Recommendation	标准已经经过广泛的技术讨论, 发布出来收集实践经验
PR	Proposed Recommendation	标准已经经过大量的实践, 委员会正在进行最后的复核
REC	Recommendation	标准已经作为推荐标准

某个 CSS 3 属性能否在特定浏览器中使用与其所在模块在 W3C 的现状并没有直接的关联关系。比如前面章节提到的 `calc` 函数能否在 Safari 浏览器里使用, 不能通过 W3C 来确认。因为 CSS 3 的发展过程并不是 W3C 先制定标准, 然后各大浏览器厂商再根据标准的进度推进实现。W3C 标准和浏览器实现是个齐头并进、相互影响的过程。

如果想知道某个 CSS 3 的属性在特定浏览器的支持情况, 可以使用网站 <http://caniuse.com/> 来查询。使用 caniuse 网站查询 `calc` 函数在浏览器中支持度的结果如图 4.4 所示。

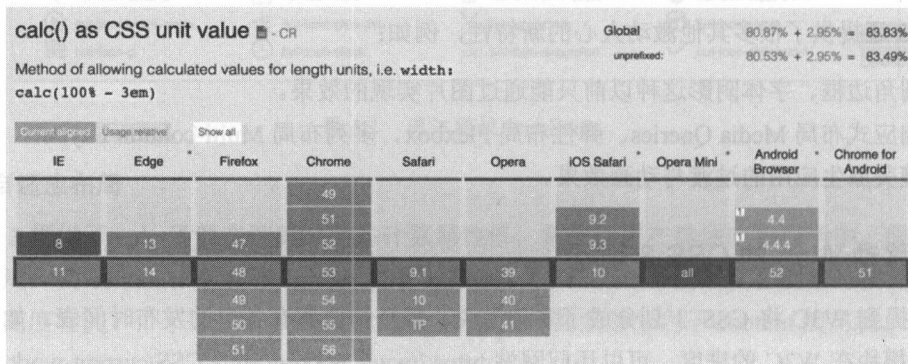


图 4.4 `calc` 函数的浏览器兼容性

在详细解读这些数据之前, 对照图 4.4 来普及几个关于浏览器版本的知识。

- iOS 上 Safari 的版本是与操作系统版本绑定的, 所以图 4.4 中“iOS Safari”列, “9.2”表示的就是 iOS 9.2。
- 在 Android 5 之前, Android 上的原生浏览器版本与操作系统绑定, 所以图 4.4 里“Android Browser”列, “4.4”表示是 Android 4.4。
- 从 Android 5 开始, Android 上的原生浏览器的版本可以单独更新, 所以不能再使用 Android 版本号来表示浏览器版本号。图 4.4 里“Android Browser”列, “52”表示 Chromium 内

核的版本。如果把鼠标悬浮其上,可以看到详细说明,比如“Android Browser”列的“52”,补充说明为 Chromium 52 是 Android 5 至 Android 6.x 的默认浏览器版本。

- 对于 Hybrid APP, 即在 APP 内部通过 WebView 组件访问 Web 页面的情况, WebView 的内核即原生浏览器的内核。
- 很多手机厂商都会为自己定制浏览器, 同一个网页在 HTC 和三星手机自带浏览器上的运行效果是可能存在差异的, 更不要说那些第三方浏览器的加入, 会使得情况变得更加复杂。查看演示稿 <http://slides.com/html5test/the-android-browser#/>, 详细描述了 Android 平台下浏览器的乱象。
- iOS 浏览器的情况相对简单。首先, 不会有厂商定制的情况, 其次, 所有第三方浏览器都被强制使用了 iOS 自带的内核。

了解了浏览器版本相关的知识之后, 现在开始解读查询数据。先看 calc 函数在各大浏览器的支持情况。如图 4.5 的箭头所指, “iOS Safari”列, 当鼠标移到“9.2”时, 出现浮层, 上面介绍此浏览器对当前属性的支持情况, 以及此浏览器的市场占有率。如果是在网页中查看, 还会发现“9.2”格子为绿色, 表示支持。

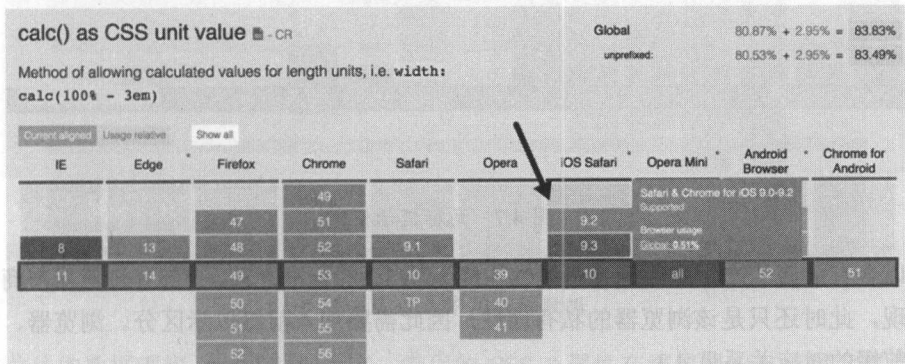


图 4.5 calc 函数的浏览器兼容性

使用同样的方式可以查看其他浏览器的支持情况。比如“IE”列, Internet Explorer 8 不支持 calc 函数, 颜色为红色。“Android Browser”列, Android 4.4 的颜色又不太一样, 为嫩绿色, 表示部分支持。同时浮层上会有说明在 Android 4.4 下, calc 函数的表达式不支持乘法和除法运算。

默认情况下只会显示部分浏览器的支持情况, 如果想知道完整的浏览器支持列表, 比如 iOS 9.1 的支持情况, 可以单击 Show all 按钮, 如图 4.6 箭头方向所示。

当显示完整的浏览器支持列表后, 如图 4.7 箭头所示, 某些项的右上角会出现角标。比如 calc 函数在 iOS 6.1 下需要添加“-webkit-”前缀才能生效, 完整的写法为“width:-webkit-calc(expression)”。

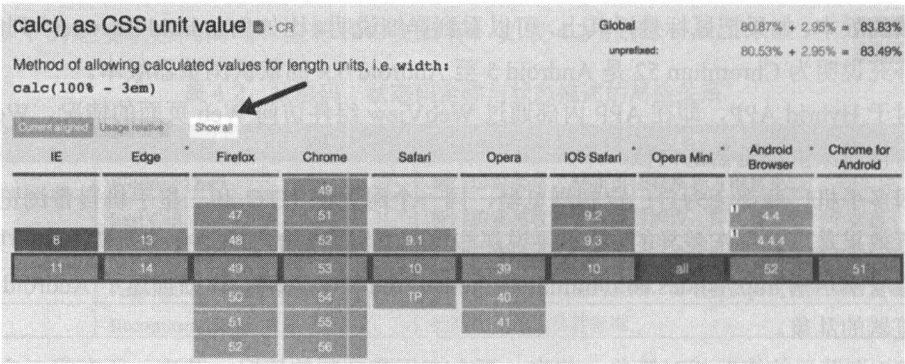


图 4.6 显示完整的浏览器支持列表

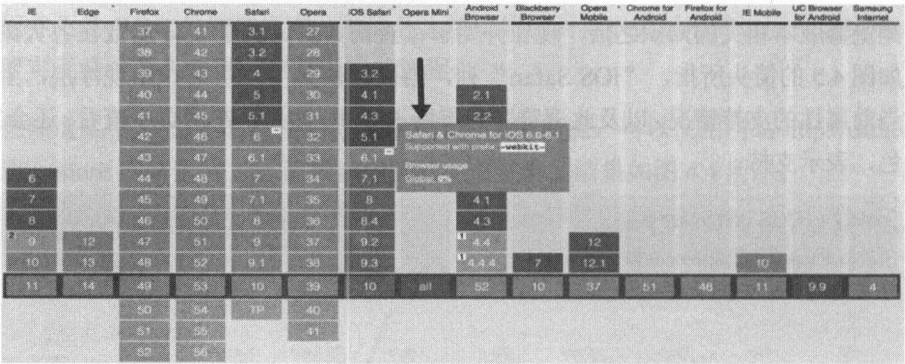


图 4.7 角标提示框

之所以会有前缀的产生是因为某个属性在成为 W3C 的标准之前,一部分浏览器产商已经先有了对应实现,此时还只是该浏览器的私有属性,因此需要加入前缀以示区分。浏览器、浏览器内核与属性前缀的对应关系见 4.3。

表 4.3 浏览器、浏览器内核与属性前缀的对应关系

属性前缀	浏览器内核	浏览器
-webkit-	WebKit	Safari, Google Chrome, Android Browser
-moz-	Gecko	Firefox
-ms-	Trident	Internet Explorer, Edge
-o-	Presto	Opera

除了可以查看特定浏览器的支持情况,以及市面上所占的份额,同时还可获得所有支持 calc 函数的浏览器所占市场比率,部分支持 calc 函数的浏览器所占市场比率。如图 4.8 箭头所示,第 1 行表示全球支持 calc 函数的浏览器占比总和为 94.08%,第 2 行表示其中非前缀占比总和为

93.87%，数据统计截至 2017 年 6 月 3 号。

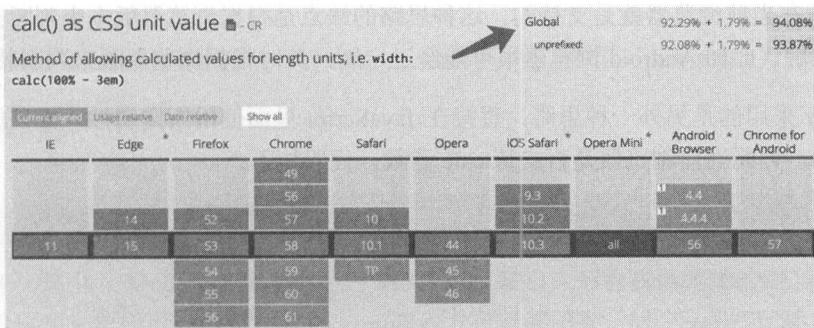


图 4.8 calc 函数所有支持的浏览器市场占比

如果想了解中国市场的数 据，如图 4.9 所示，在搜索时，caniuse 站点会根据用户所在国家，提示是否需要导入当前国家的数据。单击“Import”按钮，显示全球和当前国家的统计数据。

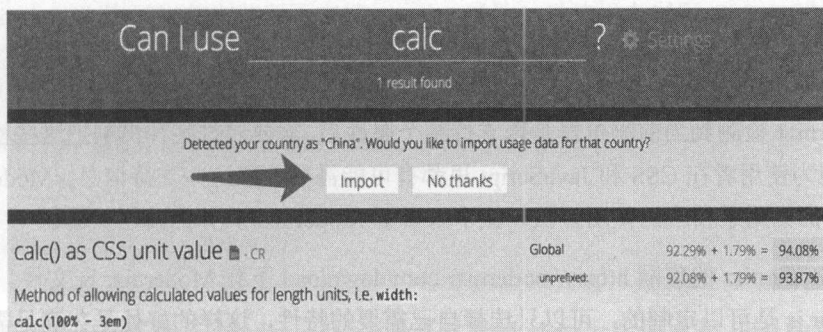


图 4.9 导入中国统计数据

通过总体的数据观察，对移动端而言，常用的 CSS 3 属性在浏览器的支持程度普遍较高。读者进行移动端开发时，应该尽量使用 CSS 3 新属性，提高开发效率及页面性能。

4.1.3 用 Modernizr 检测浏览器是否支持 CSS 3

开发者通过 caniuse 网站能在开发前了解属性在浏览器的端支持情况，同样在程序运行阶段也需要思考兼容性问题，接下来将介绍使用 Modernizr，一个用于检测用户浏览器的 HTML 5 与 CSS 3 特性的 JavaScript 库。

先看下 Modernizr 的实现原理。要实现浏览器动态特征检测，一般有如下两种思路。

第一种是使用 JavaScript 获取到 User Agent，然后根据 User Agent 判断浏览器的版本，再根据

浏览器的版本, 来确定哪些属性是在当前浏览器里支持的。比如当前浏览器如果是 iOS Safari 10, 那么 `calc` 函数在当前浏览器就是支持的。这种思路的缺点是根据浏览器版本来判断某属性是否支持并非完全准确, 比如 Android 浏览器很可能经过定制, 此时根据浏览器版来判断就可能出错。

Modernizr 采用的是另外一种思路, 直接在 JavaScript 里面使用指定属性, 如果成功执行则说明支持。比如, 判断当前浏览器是否支持 `calc` 函数, 代码如下:

```
Modernizr.addTest('csscalc', function() { // 测试是否支持 calc 函数
    var prop = 'width: ';
    var value = 'calc(10px)';
    var el = document.createElement('div'); // 创建一个 div 元素
    // 将包含了 calc 函数的 width 属性应用到创建的 div 元素上
    el.style.cssText = prop + Modernizr._prefixes.join(value + prop);
    /*
    如果 width 属性没有成功应用到创建的 div 元素上, 则 div 元素还是没有任何 style,
    说明当前浏览器不支持 calc 函数
    */
    return !!el.style.length;
});
```

当 Modernizr 检测到当前浏览器是否支持某个属性后, 需要通过某种机制把这些检测结果暴露给使用者。因为使用者在 CSS 和 JavaScript 里都有可能需要获取这些支持信息, Modernizr 给 CSS 和 JavaScript 都暴露了相应的使用方法。接下来讲解 Modernizr 具体的使用步骤。

首先去 Modernizr 的官网 <https://modernizr.com/download> 下载 Modernizr.js 文件。如图 4.10 所示, Modernizr.js 是可以定制的, 可以只选择自己需要的特性, 这样的好处是在满足需求的情况下保证 Modernizr.js 文件尽可能的最小化。

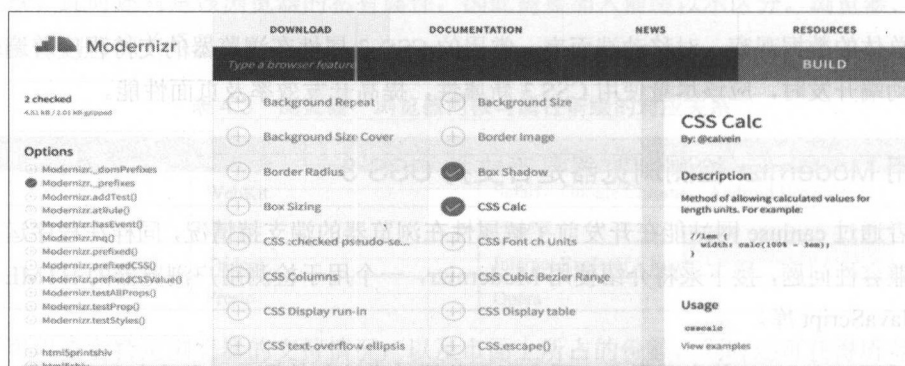


图 4.10 下载可定制的 Modernizr.js

下载了 Modernizr.js 以后, 将其插入 HTML 页面的头部 Head 标签中。代码如下:

```
<!DOCTYPE html>
<html class="no-js" lang="en">      // 当引入 Modernizr.js 后, no-js 会被修改为 js
<head>
  <meta charset="utf-8">
  <script src="js/modernizr.js"></script> // 在 head 标签中引入 Modernizr.js 文件
</head>
</html>
```

之所以在头部引入 Modernizr.js, 是因为 Modernizr 被引入后会立即开始检测当前浏览器支持的属性, 并将检测结果通过 CSS 和 JavaScript 两种方式暴露出来。Modernizr 会将 HTML 标签元素的样式修改成类似如下代码:

```
<html class="js no-opacity postmessage history csscalc boxshadow">
```

Modernizr 添加了“csscalc”这个样式类名到 HTML 标签中, 表示当前浏览器支持 calc 函数。反之, “no-opacity”表示当前浏览器不支持 opacity 属性。当使用者书写 CSS 的时候, 可以根据某个属性是否支持而书写不同的样式。代码如下:

```
.no-csscalc {
  // 如果当前浏览器不支持 calc 函数的使用
}
.csscalc {
  // 如果当前浏览器支持 calc 函数的使用
}
```

除了添加对应的样式类名, Modernizr 还会暴露对象 Modernizr 给使用者, 并且给对象 Modernizr 添加对应的布尔型属性, 代码如下:

```
if (Modernizr.csscalc) {
  // 如果当前浏览器支持 calc 函数的使用
} else {
  // 如果当前浏览器不支持 calc 函数的使用
}
```

如果想知道某个 CSS 属性在 Modernizr 里对应暴露的 CSS 类名和 JavaScript 对象名, 可以访问站点 <https://modernizr.com/docs>。“Features detected by Modernizr”表格里记录了详细的映射关系, 如图 4.11 所示。

到此, Modernizr 的基本使用方式和背后的设计原理已经讲述完成。有关 Modernizr 的完整信息可以上官网 <https://modernizr.com/> 查询。

Features detected by Modernizr	
Detect	CSS class/JS property
Ambient Light Events	<code>ambientlight</code>
Detects support for the API that provides information about the ambient light levels, as detected by the device's light detector, in terms of lux units.	
Application Cache	<code>applicationcache</code>
Detects support for the Application Cache, for storing data to enable web-based applications run offline. The API has been heavily criticized and discussions are underway to address this.	
HTML5 Audio Element	<code>audio</code>
Detects the audio element	

图 4.11 Modernizr 检测特性与暴露名称的映射关系

最后着重讲述下 Modernizr 提供的另外一个重要功能，自定义特征检测。比如检测当前浏览器是否为 Internet Explorer 7，Modernizr 原生并没有提供，但允许用户自定义特征检测函数，代码如下：

```
Modernizr.addTest('ie7', function() {  
    if (current browser is Internet Explorer 7) {  
        return true;  
    }  
    else {  
        return false;  
    }  
});
```

当浏览器是 Internet Explorer 7 时，Modernizr 会添加 “ie7” 样式类名到 HTML 元素，并且在 JavaScript 里面调用 `Modernizr.ie7` 返回 `true`。当浏览器不是 Internet Explorer 7 时，Modernizr 会添加 “no-ie7” 这个样式类名到 HTML 元素，并且在 JavaScript 里面调用 `Modernizr.ie7` 返回 `false`。

4.2 选择器

选择器（Selectors）是链接文档中的元素和 CSS 样式的桥梁，通过选择器可以为指定的元素应用样式。虽然浏览器提供了丰富的选择器，但各浏览器的支持情况并不统一，选择器也存在兼容问题。本章将先会介绍常见的选择器，之后再重点介绍伪类和伪元素选择器，及选择器的优先级和权重，最后通过实战应用本章学习到的内容。

4.2.1 常见选择器

1. 选择器介绍

CSS 样式表中定义一条 CSS 样式的格式如下：

选择器 { 样式 }

选择器指定了随后的样式将会应用到文档中的哪些元素上。首先，在学习高级选择器使用之前，来了解一下 CSS 中一些常见选择器，基本选择器和选择器分组，及选择器对应的 CSS 版本，见表 4.4。

表 4.4 基本选择器和选择器分组

选 择 器	名 称	版 本
*	通用选择器 (Universal selectors)	2.1
E	标签选择器 (Type selectors)	1
.class	类选择器 (Class selectors)	1
#id	ID选择器 (ID selectors)	1
E[attr]	属性选择器 (Attribute selectors)	2.1
E[attr=val]	属性选择器 (Attribute selectors)	2.1
E[attr~=val]	属性选择器 (Attribute selectors)	2.1
E[attr =val]	属性选择器 (Attribute selectors)	2.1
E[attr^=val]	属性选择器 (Attribute selectors)	3
E[attr\$=val]	属性选择器 (Attribute selectors)	3
E[attr*=val]	属性选择器 (Attribute selectors)	3
E F	后代选择器 (Descendant selectors)	1
E > F	子选择器 (Child selectors)	2.1
E + F	相邻兄弟选择器 (Adjacent sibling selectors)	2.1
E ~ F	兄弟选择器 (General sibling selectors)	3
S1, S2, ...	选择器分组 (Groups of selectors)	1

提示：选择器分组是由以逗号分隔的多个选择器组合而成，使用选择器分组，可以为多个具有相同属性的元素定义样式，这样做可以极大地简化 CSS 样式代码，对于减少 CSS 文件体积是一个好的办法。

基本选择器和选择器分组的使用方法，见表 4.5。

表 4.5 基本选择器和选择器分组的使用实例

选 择 器	实 例	描 述
*	*	匹配所有的元素
E	p	匹配所有的<p>元素

选择器	实 例	描 述
.class	.nav	匹配所有class="nav"的元素
#id	#wrapper	匹配所有id="wrapper"的元素
E[attr]	a[data-url]	匹配所有带有"data-url"属性的<a>元素
E[attr=val]	input[type="text"]	匹配所有带有type="text"属性的<input>元素
E[attr~val]	div[keywords~="round"]	匹配所有"keywords"属性中包含"round"的<div>元素
E[attr =val]	label[lang="zh"]	匹配所有"lang"属性以"zh-"开头的<label>元素
E[attr^=val]	a[href^="http://"]	匹配所有"href"属性以http://开头的<a>元素
E[attr\$=val]	img[src\$=".png"]	匹配所有"src"属性以".com"结尾的元素
E[attr*=val]	a[href*="hujiang.com"]	匹配所有href属性中包含"hujiang.com"的<a>元素
E F	.blog p	匹配所有class="blog"元素的后代<p>元素
E > F	.nav > button	匹配所有class="nav"元素的子<button>元素
E + F	header ~ div	匹配<header>后面的所有同级<div>元素
E ~ F	label + input	匹配所有紧跟在<label>元素之后的<input>元素
S1, S2, ...	input, select, textarea	匹配所有的<input>、<select>、<textarea>元素

2. 选择器使用综合实例

下面通过实例来了解常见选择器和选择器分组的使用。本例实现一个简单的登录界面，包含标题、用户名和密码输入框和登录按钮，代码如下：

```

01 <head>
02   <style>
03     * { margin: 0; padding: 0; }           /* 外边距、内边距 */
04     body { background: #F0F0F0; }          /* 背景颜色 */
05     header {
06       height: 48px;                        /* 高度 */
07       line-height: 48px;                  /* 行高 */
08       background: #2e353d;                /* 背景色 */
09       color: white;                       /* 字体颜色 */
10       font-size: 16px;                    /* 字号 */
11       text-align: center;                 /* 文字水平居中 */
12     }
13     header+#content { margin-top: 20px } /* 上边距 */
14     #content { padding: 10px; }
15     input, button { width: 100%; }
16     input[type="text"], input[type="password"] {
17       height: 35px;
18       margin-top: 10px;
19     }

```

```

20     button { height: 30px; margin-top: 20px; }
21   </style>
22 </head>
23 <body>
24   <header>用户登录</header>           <!-- 标题 -->
25   <div id="content">                 <!-- 内容容器 -->
26     <input type="text" value="tom"><br> <!-- 用户名输入框 -->
27     <input type="password" value="123456"> <!-- 密码输入框 -->
28     <button>登录</button>           <!-- 登录按钮 -->
29   </div>
30 </body>

```

运行效果如图 4.12 所示。

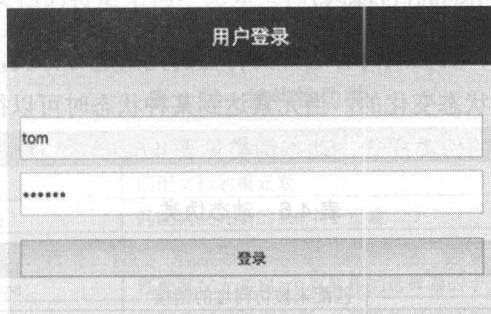


图 4.12 运行结果

例子中使用的选择器如下。

- 通用选择器：“*”为文档中的全部元素。
- 标签选择器：“body”、“header”和“button”，分别选择内容、标题和按钮。
- 相邻兄弟选择器：“header+#content”为紧接在标题元素后面的内容容器。
- ID 选择器：“#content”为内容容器。
- 选择器分组：“input,button”为所有 Input 和 Button 元素。
- 属性选择器：“input[type=“text”],input[type=“password”]”为包含属性“type”，并且属性 type 为“text”或“password”的 Input 元素。

本例只是一个简单的应用场景，在复杂场景下，需要更灵活地使用选择器。

4.2.2 伪类和伪元素

CSS 选择器除了基本选择器和选择器分组外，还包括伪类选择器和伪元素选择器。

1. 伪类

伪类 (Pseudo-classes) 用来指定选择器的某种特定状态或者条件, 伪类在 DOM 中并不存在, 但对用户却是可见的。比如 A 元素, 可以通过伪类指定未单击状态和已单击状态。伪类选择器的基本语法如下:

```
selector:pseudo-classes{ property: value }
```

例如, 给所有访问过的超链接设置字体颜色为红色, 代码如下:

```
a:visited { color: red }
```

CCS 中的伪类有以下种类:

- 动态伪类 (Dynamic pseudo-classes)

动态伪类对除了其名称、属性或内容之外的特征的元素进行分类, 不会显示在文档源或文档树中。动态伪类是随元素的状态变化的, 当元素达到某种状态时可以得到某种伪类样式, 失去这种状态时, 也会失去相对应的伪类样式, 动态伪类见表 4.6。

表 4.6 动态伪类

选 择 器	实 例	描 述	版 本
:link	a:link	匹配未被访问过的链接	1
:visited	a:visited	匹配被访问过的链接	1
:hover	a:hover	匹配鼠标指针在其上浮动的元素	1
:active	a:active	匹配鼠标指针在其上按下的元素	1
:focus	input:focus	匹配获得焦点的元素	2.1

- 目标伪类 (The target pseudo-class)

目标伪类指定当前活动的锚, 使用目标伪类可以为活动的锚设置样式, 目标伪类见表 4.7。

表 4.7 目标伪类

选 择 器	实 例	描 述	版 本
:target	#tbl1:target	匹配活动的锚	3

- 语言伪类 (The language pseudo-class)

语言伪类向带有指定 lang 属性的元素添加样式, 语言伪类见表 4.8。

表 4.8 语言伪类

选 择 器	实 例	描 述	版 本
:lang(val)	p:lang(en)	匹配带有指定 lang 属性的元素	3

- UI 元素状态伪类 (The UI element states pseudo-classes)

UI 元素状态伪类主要用来指定表单中的元素状态, UI 元素状态伪类见表 4.9。

表 4.9 UI 元素状态伪类

选 择 器	实 例	描 述	版 本
:enabled	input:enabled	匹配启用的元素	3
:disabled	input:disabled	匹配禁用的元素	3
:checked	input:checked	匹配被选中的元素	3

注意: display 和 visibility 属性对于 UI 元素状态伪类匹配 enabled/disabled 状态没有影响。

- 结构性伪类 (Structural pseudo-classes)

结构性伪类用来指定文档的特定结构, 比如文档根元素、某元素的第一个子元素等。结构性伪类见表 4.10。

表 4.10 结构性伪类

选 择 器	实 例	描 述	版 本
:root	:root	匹配文档的根元素	3
:nth-child(n)	:nth-child	匹配其父元素的第 n 个子元素	3
:nth-last-child(n)	:nth-last-child	匹配其父元素倒数第 n 个子元素	3
:nth-of-type(n)	:nth-of-type	匹配其父元素第 n 个有着相同选择器的子元素	3
:nth-last-of-type(n)	:nth-last-of-type	匹配其父元素倒数第 n 个有着相同选择器的子元素	3
:first-child	:first-child	匹配其父元素的第一个子元素	3
:last-child	:last-child	匹配其父元素的最后一个子元素	3
:first-of-type	:first-of-type	匹配其父元素第一个有着相同选择器的子元素	3
:last-of-type	:last-of-type	匹配其父元素最后一个有着相同选择器的子元素	3
:only-child	:only-child	匹配其父元素的唯一子元素	3
:only-of-type	:only-of-type	匹配其父元素的唯一有着相同选择器的子元素	3
:empty	:empty	匹配没有子元素 (包括文字节点) 的元素	3

:nth-child(n)、:nth-last-child(n)、:nth-of-type(n)、:nth-last-of-type(n) 中的 n , 也可以写成表达式“ $an+b$ ”, 其中, a 和 b 是 0 或正整数, 如 $2n$ 、 $2n+1$ 、 $4n-1$ 等。

n 是从 0 开始的整数, 表达式的写法相当于把每 a 个子元素分成一组, 取每组的第 b 个元素。比如 $2n$ 表示匹配所有顺序是 2 的倍数的子元素。 $2n$ 和 $2n+1$ 也可以分别写成 even 和 odd, 表示第偶数、奇数个子元素。

:nth-child(n) 和 :nth-of-type(n) 的区别是, :nth-child(n) 表示其父元素的第 n 个子元素, 无论其子元素的类别是什么, 而 :nth-of-type(n) 表示的是同类别的第 n 个子元素, 比如列表。实例代码如下:


```
<ul>
  <span>列表标题</span>
  <li>列表项目一</li>
  <li>列表项目二</li>
  <li>列表项目三</li>
</ul>
```

选择器 `li:nth-child(2)`，匹配的是 UL 元素的第二个子元素，并且这个元素是 LI，即“列表项目一”。而选择器 `li:nth-of-type(2)`，取的是 UL 的第二个 LI 子元素“列表项目二”。

• 否定伪类

否定伪类是用来选择所有非指定类型元素的其他元素，见表 4.11。

表 4.11 否定伪类

选 择 器	实 例	描 述	版 本
<code>:not(s)</code>	<code>input:not([type="text"])</code>	匹配所有非指定类型的其他元素	3

2. 伪元素

伪元素（Pseudo-elements）是指不存在于文档树中的抽象内容，比如某元素内容的第一个字母或第一行。伪元素选择器的基本语法如下：

```
selector::pseudo-elements{ property: value }
```

例如，给文本的第一个字母设置颜色为红色，代码如下：

```
p::first-letter { color: red }
```

伪元素见表 4.12。

表 4.12 伪元素

选 择 器	实 例	描 述	版 本
<code>::first-line</code>	<code>p::first-line</code>	匹配元素文本内容的首行	1
<code>::first-letter</code>	<code>p::first-letter</code>	匹配元素文本内容的首个字母	1
<code>::before</code>	<code>div::before</code>	元素之前	2.1
<code>::after</code>	<code>span::after</code>	元素之后	2.1

注意：在 CSS 1 和 CSS 2 中，伪元素选择器中只有一个“:”，而在 CSS 3 中变成两个，即由“`:first-line`”变成“`::first-line`”，这个变化主要是考虑把伪类和伪元素区分开。目前两种格式的写法都是被接受的，效果也是一样的。

下面通过实例了解伪元素的使用，代码如下：

```

01 <head>
02   <style>
03     span::before {
04       display: inline-block;          /* 显示模式 */
05       width: 4px;                    /* 宽度 */
06       height: 14px;                 /* 高度 */
07       margin-right: 10px;            /* 右外边距 */
08       background: #198be5;           /* 背景颜色 */
09       content: "";                   /* 内容 */
10     }
11     p::first-letter { color: red; }   /* 文字红色 */
12     p::first-line { font-style: italic; } /* 文字倾斜 */
13     p::after {
14       display: block;
15       height: 1px;
16       content: "...";
17     }
18   </style>
19 </head>
20 <body>
21   <span>Pseudo-elements</span>      <!-- 标题 -->
22   <p>                                <!-- 段落 -->
23   Pseudo-elements create abstractions about the document tree beyond those specified by the
24   document language.
25   </p>
26 </body>

```

运行效果如图 4.13 所示。

■ Pseudo-elements

Pseudo-elements create abstractions about the document tree beyond those specified by the document language.

...

图 4.13 伪元素实例运行结果

提示：使用::before 和::after 两个伪元素时，content 属性必须设置，否则伪元素不能生效，如果不想显示 content 内容，可以设置值为空引用。

上面的例子中, 运用了前面介绍的伪元素选择器“`span::before`”, 为标题前端添加了一个蓝色小区块。选择器“`p::first-letter`”, 为段落的第一个字母设置了文字红色。选择器“`p::first-line`”, 为段落的第一行文字设置斜体样式。选择器“`p::after`”, 为段落的后面添加了“...”省略符号。

4.2.3 优先级和权重

本节讲解 CSS 优先级和权重的概念, 权重决定了 CSS 规则怎样被浏览器解析直到生效, 当很多条规则被应用到某个元素上时, 最终呈现的效果是按照规则各自的优先级和权重来决定的, 先来看一个简单的例子, 代码如下:

```
01 <style type="text/css">
02     .red{color: red;}
03     .blue{color: blue;}
04 </style>
05 <div class="wrapper">
06     <div class="content">
07         <p id="hello" class="blue red">我爱 CSS 3</p> <!-- 显示为蓝色 -->
08     </div>
09 </div>
```

例子中的“我爱 CSS 3”最终显示为蓝色, 这是因为这两种不同的颜色规则拥有相同的权重。而在 CSS 中, 后面定义的规则会覆盖前面与之相同属性的规则, 这里所指的“后面”, 并非 P 标签中 class 属性的顺序, 而是指 Style 标签中 CSS 代码中定义的规则顺序。接下来稍做修改, 代码如下:

```
p.red{color: red;}
.blue{color: blue;}
```

刷新页面以后发现“我爱 CSS 3”变成红色, 也就是说定义在前面的规则优先发挥了作用。在解释其中原因之前, 先来看一下 CSS 中权重的划分, CSS 中的权重分为 4 个级别, 如下:

- 内联样式 (HTML 文档中的 style 属性)
- ID 选择器
- 类、伪类, 属性选择器
- 元素、伪元素

这 4 个级别的权重从高到低代表不同的优先级, 内联样式的优先级最高, 而元素和伪元素的优先级最低, 最终权重值根据不同的优先级加权计算。可以使用“0, 0, 0, 0”这样来表示一个权重, 分别对应上面的 4 个级别。

现在,回过头来计算上面例子的权重,“p.red”的权重为一个元素选择器和一个类选择器表示为“0, 0, 1, 1”。“blue”的权重为一个类选择器表示为“0, 0, 1, 0”。在比较权重小时,先从高优先级的权重开始比较,遇到相同值后再比较后面的优先级。因为“p.red”多了一个元素选择器,因此也就拥有更高的权重。浏览器在处理相同权重的规则时,定义在后面的规则会生效,而在处理不同权重的规则时,则是权重更高的规则生效。

提示:在有些参考文案中会将这4个级别的权重值描述成“1000, 100, 10, 1”这样的数字,但是需要注意的是,在做权重计算时,低级别的权重不会进位成高级别的权重,如11个类选择器的权重并不会超过1个ID选择器。

接下来尝试计算选择器权重,见表4.13。

表 4.13 选择器权重实例

选择器/内联样式	权重说明
*	通配符和继承的属性,权重为0, 0, 0, 0
ul li	2个元素选择器,权重为0, 0, 0, 2
ul li.item	2个元素选择器, 1个类选择器,权重为0, 0, 1, 2
ul li:hover	2个元素选择器, 1个伪类选择器,权重为0, 0, 1, 2
#content	1个ID选择器,权重为0, 1, 0, 0
body #content	1个ID选择器, 1个元素选择器,权重为0, 1, 0, 1
div#content	1个元素选择器, 1个ID选择器,权重为0, 1, 0, 1
div[id="content"]	1个元素选择器, 1个属性选择器,权重为0, 0, 1, 1
div + *[rel="section"]	1个元素选择器, 1个属性选择器,权重为0, 0, 1, 1
li:first-child	1个元素选择器, 1个伪元素选择器,权重为0, 0, 0, 2
style=""	内联样式属性,权重为1, 0, 0, 0

最后,还有一个特殊的规则就是“!important”。“!important”写在CSS规则后面,可以将对应的规则提升到最高的权重,通常使用“!important”并不是一个很好的做法,因为当需要覆盖使用“!important”声明的规则时,唯一的做法是在此规则后面再次定义一个“!important”规则,这样会导致更多的混乱,使代码难以维护。

合理的规划CSS选择器权重是完成一个易于维护项目的良好开端,也是真正掌握CSS技术最基本的要求。

提示:由于浏览器在解析选择器的时候是按照从右到左的顺序进行的,这就导致更多层级选择器嵌套规则在查找时会花费更多的时间。比如“#main.class_a.class_b ul li”,

在解析规则的过程中，首先查找到的是 LI 元素而不是 ID 为“main”的元素，在从右往左匹配的过程中，会有很多规则匹配花费在失败的查找上，因此用更简短、更容易被查找到选择器是一个好的习惯。

4.3 响应式开发

传统 PC 页面的开发中，通常情况是由视觉针对一定的分辨率设计视觉稿，开发只需要实现特定的分辨率即可。随着技术的发展，移动端屏幕尺寸越来越大，并且屏幕的像素越来越高，传统的开发方式面临着多分辨率适配的问题。本节介绍的响应式开发方式提供了一种有效的解决方案。

4.3.1 常见设备的宽高

响应式开发的本质是针对多种屏幕做适配，在实际的开发中，通常情况下是对主流的设备适配。

首先，需要掌握几个基本概念。

- 物理像素：设备的屏幕实际像素点，如常说的 iPhone 6 Plus 的分辨率是 1920 像素乘以 1080 像素。
- 设备独立像素：逻辑像素，用于定义应用的 UI。
- 屏幕像素比（devicePixelRatio）：物理像素与设备独立像素的比值。

提示：UI 即用户界面，这里指的是定义应用的界面各元素的大小。

表 4.14 常见设备宽高

设备名称	物理像素	设备独立像素	屏幕像素比
iPhone 7, 6, 6S	750×1334	375×667	2
iPhone 7 Plus, 6S Plus, 6 Plus	1080×1920	414×736	3
iPhone 5, 5S, 5C, SE	640×1136	320×568	2
iPhone 4	640×960	320×480	2
iPod Touch	640×1136	320×480	2
Galaxy S7, S7 edge, S6	1440×2560	360×640	4
Galaxy S5, S4	1080×1920	360×640	3
Galaxy S4 mini	540×960	360×640	1.5
Galaxy S3	720×1280	360×640	2
Galaxy Note 4	1440×2560	360×640	4
Galaxy Note 3	1080×1920	360×640	3

续表

设备名称	物理像素	设备独立像素	屏幕像素比
Galaxy Note 2	720×1280	360×640	2
Mi 4, 3	1080×920	360×640	3
HTC One	1080×1920	360×640	3
Sony Xperia Z3	1080×1920	360×640	3
Lenovo K900	1080×1920	360×640	3
ZTE Grand S	1080×1920	360×640	3
iPad Pro	2048×2732	1024×1366	2
iPad 3, 4, Air, Air2	1536×2048	768×1024	2
iPad mini 2, 3	1536×2048	768×1024	2
iPad mini	768×1024	768×1024	1

通过表 4.14 的观察得出, 尽管物理分辨率不相同, 但是设备的独立像素是相同的。

先看一个简单的例子, 代码如下:

```
<html>
<body>
  <div style="width:320px;height:320px;background-color:gray;"></div>
</body>
</html>
```

在浏览器中的结果如图 4.14 所示。

可以看出, 屏幕的宽度是 320 像素, CSS 设置的 DIV 的宽度是 320 像素, 但是灰色的 DIV 并没有铺满整个屏幕。这是因为在 iPhone 中, 视口 (Viewport) 的默认宽度是 980 像素。

修改上面的例子, 在 Head 元素中添加如下代码:

```
<head>
  <!-- 将 viewport 的宽度设置为设备的宽度 -->
  <!-- initial-scale, 初始缩放比率。设置为 1 表示初始不缩放 -->
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
```

运行效果如图 4.15 所示。

可以看到灰色的 DIV 铺满了整个屏幕, 也就是说其宽度现在就是设备的独立像素宽度 320 像素。

在上面的实例中, 采用了 viewport 属性。viewport 是指屏幕上能用来显示网页的区域, 默认情况下大多数设备的 viewport 的宽度都是 980 像素。通过在 Head 元素中增加 Meta 标签来设置 viewport 属性。

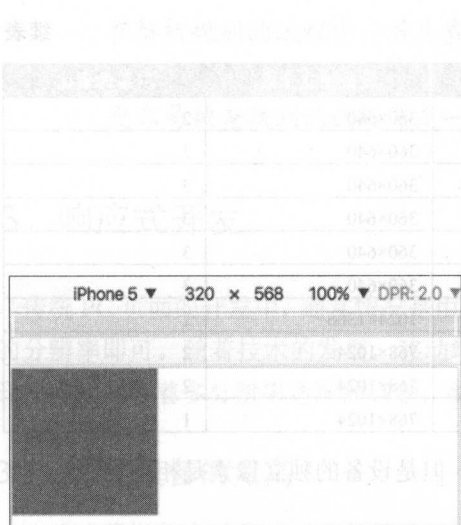


图 4.14 DIV 在 iPhone 5 下的展示效果

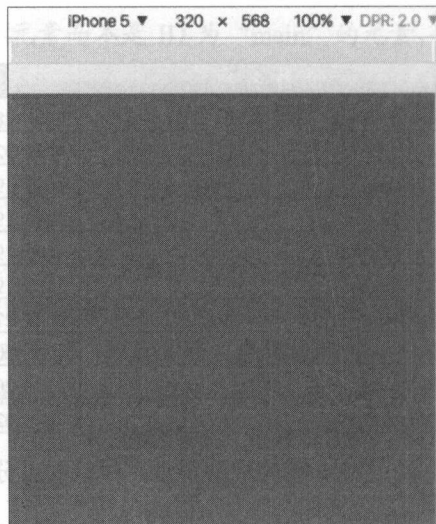


图 4.15 设置 viewport

Meta 标签包含的属性如下。

- **width**: 设置 viewport 的宽度, 为正整数, 或者字符串“device-width”。
- **initial-scale**: 设置 viewport 的初始缩放值, 为数字, 可以带小数。
- **minimum-scale**: 设置 viewport 的最小缩放值, 为数字, 可以带小数。
- **maximum-scale**: 设置 viewport 的最大缩放值, 为数字, 可以带小数。
- **height**: 设置 viewport 的高度, 该属性一般不会用到。
- **user-scalable**: 是否允许用户缩放, 值为“yes”或“no”。

通过设置 viewport 属性, 可以调整用户界面的逻辑大小, 页面 CSS 中的大小均以 viewport 为基准。如果将 viewport 的 width 设置成一个固定值, 那么在所有设备上看到的界面效果, 在水平方向上是一致的 (各元素的逻辑宽度一致)。这比较适合全屏广告专题页面的开发场景。通过 viewport, 使得开发者不必过多关心设备的物理分辨率, 降低了适配不同尺寸屏幕的难度并减少了开发成本。

4.3.2 Flex 弹性盒布局

在 Flex 出现之前, 布局基于盒模型, 依赖 display、position 和 float 样式属性。但是使用时需要注意清除浮动, 并且对于一些特定布局的实现非常不便。在下面的例子中, 里层的灰色方块需要相对于父容器垂直居中, 如图 4.16 所示。

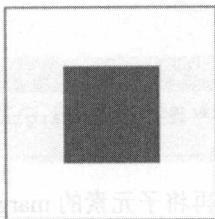


图 4.16 垂直居中

采用传统的实现方式，代码如下：

```

01 .container {
02     position: relative;
03     height: 90vw;
04     width: 90vw;
05     margin: 0; border: 1px solid gray;
06 }
07 box {
08     position: absolute;           /* 相对于父容器绝对定位 */
09     top: 45vw;                   /* top 设置为父容器高度的一半 */
10     width: 60px; height: 60px;
11     left: 45vw;                 /* left 设置为父容器宽度的一半 */
12     margin-top: -30px;           /* margin-top 设置为自己高度的负一半 */
13     margin-left: -30px;         /* margin-left 设置为自己宽度的负一半 */
14     background-color: gray;
15 }

```

提示：在这个例子中，采用了 vw 单位，1vw 指的是当前页面 viewport 的宽度的 1/100。

先设置父容器的样式 position 值为 relative，以便于子元素相对父容器定位。然后设置子元素的样式 left 和 top 为父容器宽度和高度的一半，此时子元素的左上顶点在父容器的中心。然后通过设置样式 margin 分别为高宽的一半负值，将子元素的位置移回居中。该实现方式存在一个明显的缺点，父容器和子容器的宽度需要固定。如果子元素中的高度随内容而变化，高度不固定，就无法实现垂直居中对齐。而采用 Flex 布局，可以很轻松地实现该效果。代码如下：

```

01 .container{
02     height: 200px;
03     display: flex;               /* 设置父容器为 flex 布局 */
04     border: 1px solid gray;
05 }
06 .box{

```



```
07     width: 60px;
08     height: 60px;
09     background-color: gray;
10     margin: auto;           /* 设置子项的 margin 未 auto 时，会自动居中 */
11 }
```

对父容器 `display` 属性设置为 `flex`，再将子元素的 `margin` 属性设置为 `auto`，就可以实现子元素相对父元素居中。这种方式代码非常简洁，并且父容器的宽度不需要固定，父容器的宽度发生变化，灰色方块始终保持居中。

注意：采用 `Flex` 布局后，子元素的样式 `float`、`clear` 和 `vertical-align` 失效。

上述例子，简单地介绍了采用 `Flex` 布局的便利性，下面介绍该布局的基本概念。`Flex` 布局的结构如图 4.17 所示。

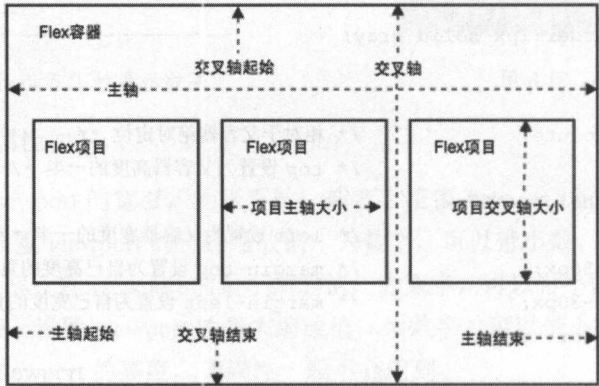


图 4.17 Flex 布局结构

采用 `Flex` 布局的容器称为 `Flex` 容器，所有子元素自动成为容器成员，称为 `Flex` 项目。容器默认存在两根轴，水平的主轴和垂直的交叉轴。主轴的开始位置叫作“主轴起始”，结束位置叫作“主轴结束”。交叉轴的开始位置叫作“交叉轴起始”，交叉轴的结束位置叫作“交叉轴结束”。项目默认沿主轴排列，单个项目占据主轴的空间叫作“项目主轴大小”，占据交叉轴的空间叫作“项目交叉轴大小”。

`Flex` 容器上包含多种属性，见表 4.15。

表 4.15 Flex 容器属性

属性名称	说明	可选值
<code>flex-direction</code>	决定主轴的方向（项目的排列方向）	<code>row</code> 、 <code>row-reverse</code> 、 <code>column</code> 、 <code>column-reverse</code>

续表

属性名称	说明	可选值
flex-wrap	默认情况下, 项目都排在轴线上, 如果超出一行, 该属性定义如何换行	nowrap、wrap、wrap-reverse
justify-content	定义项目在主轴上的对齐方式	flex-start、flex-end、center、space-between、space-around
align-items	定义项目在交叉轴上的对齐方式	flex-start、flex-end、center、baseline、stretch
align-content	定义多根轴线的对齐方式, 如果项目只有一根轴线, 该属性不起作用	flex-start、flex-end、center、space-between、space-around、stretch

Flex 项目包含的属性见表 4.16。

表 4.16 Flex 项目属性

属性名称	说明
order	定义项目的排列顺序, 值越小, 排列越靠前, 默认值为0
flex-grow	定义项目的放大比例, 默认为0
flex-shrink	定义项目的缩小比例, 默认为1
flex-basis	定义了分配剩余空间之前, 项目占据的主轴空间。浏览器根据这个属性值, 计算主轴的剩余空间。默认值为“auto”, 即项目的实际大小
align-self	设置单个项目的对齐方式, 可覆盖Flex容器设置的align-items属性。可选值auto、flex-start、flex-end、center、baseline、stretch

下面通过实例介绍 Flex 布局的使用, 通常在项目中, 页面底部导航菜单项需要等宽显示, 如图 4.18 所示。



图 4.18 底部导航菜单

要实现如图 4.18 所示的导航, 需要设置容器为 Flex 布局, 并且设置每一个子元素的样式 flex 属性值为 1。HTML 部分代码如下:

```
<footer>
  <a href="#"><span class="icon-home"></span><p>首页</p></a>
  <a href="#"><span class="icon-delivery"></span><p>物流</p></a>
  <a href="#"><span class="icon-cart"></span><p>购物车</p></a>
  <a href="#"><span class="icon-account"></span><p>我的账户</p></a>
  <a href="#"><span class="icon-more"></span><p>更多</p></a>
</footer>
```

CSS 部分代码如下:

```

01 footer{
02     border-top: 1px solid #e7e7e7;
03     border-bottom: 1px solid #f8f8f8;
04     display: -webkit-box;           /* 为 webkit 适配, 同 display: flex */
05     display: -moz-box;             /* 为 Firefox 适配, 同 display: flex */
06     display: -ms-flexbox;          /* 为 Internet Explorer 适配, 同 display: flex */
07     display: flex;                 /* 设置容器为 Flex 布局 */
08     -webkit-box-pack: justify;      /* 为 webkit 适配, 平均分配水平空间给每个元素 */
09     -moz-box-pack: justify;         /* 为 Firefox 适配, 代码意义同上 */
10     -ms-flex-pack: justify;         /* 为 Internet Explorer 适配, 代码意义同上 */
11     -webkit-box-align: center;      /* 为 webkit 适配, 项目垂直居中 */
12     -moz-box-align: justify;        /* 为 Firefox 适配, 代码意义同上 */
13     -ms-flex-align: center;         /* 为 Internet Explorer 适配, 代码意义同上 */
14     justify-content: space-between; /* 项目平均分布在一行 */
15     align-content: center;          /* 项目垂直居中 */
16     padding-top: 0.5em;
17 }
18 footer a{
19     -webkit-box-flex: 1;            /* 为 webkit 适配, 每一项目等宽 */
20     -moz-box-flex: 1;               /* 为 Firefox 适配, 意义同上 */
21     -webkit-flex: 1;                /* 为 webkit 适配, 意义同上 */
22     -ms-flex: 1;                    /* 为 Internet Explorer 适配, 意义同上 */
23     flex: 1;                        /* 每一项目的宽度相同 */
24     text-align: center;
25     text-decoration: none;
26     color: #5d656b;
27 }
28 footer a span{font-family: iconfont; font-size: 1.5em;}
29 footer a p{font-size: 10px; margin: 0;}

```

在上面的例子中, 可在处理 Flex 布局时, 额外添加代码来应对不同浏览器的兼容性, 因为在 Flex 支持的初期, 各大浏览器支持的标准不一致。不过, 目前新版的浏览器都已支持规范的 Flex 标准。

4.3.3 媒体查询 (Media Query)

设备的尺寸和分辨率千差万别, 尤其是移动设备, 屏幕的分辨率变得更复杂。分辨率的差异化扩大, 使得不得不根据屏幕的分辨率给用户呈现不同的界面。媒体查询的出现, 有效地解决了这一问题。

在沪江移动站中,设计师针对移动端的设备在小屏幕的情况下,页面展开铺满屏幕,效果如图 4.19 所示。



图 4.19 小屏幕尺寸

在大屏幕的情况下,需要固定主体内容的宽度,并居中显示,如图 4.20 所示。



图 4.20 大屏幕尺寸

对于这种情况,采用媒体查询就可以依据不同的设备大小,使用不同的 CSS 样式,代码如下:

```
01 .wrapper {
02     padding-top: 50px;                /* 宽度不限制 */
03     background: #FFF;
04 }
05 @media screen and (min-width: 640px) { /* 在屏幕上,并且分辨率宽度不小于 640px */
06     .wrapper {                        /* 固定宽度,并且通过 margin 水平居中 */
07         width: 480px;
08         margin: 0 auto;
09     }
10 }
```

CSS 代码由上往下解析,默认情况下,适配代码第 1 行样式类“.wrapper”。当满足媒体查询条件的情况下,适配代码第 5 行媒体查询中定义的样式。

注意: 在使用媒体查询的时候, 需要注意样式先后顺序, CSS 由上往下解析。

媒体查询有两种用法, 一个是定义在 CSS 中, 如上述的例子。另一种是直接定义在页面的 Link 元素, 采用 `media` 属性, 代码如下:

```
<link rel="stylesheet" media="(max-width: 800px)" href="example.css" />
```

在这种情况下, 引用的 CSS 资源始终会加载, 并由 `media` 属性中定义的条件来决定引用的样式是否生效。

媒体查询的条件可以是多个, 采用 “,” 隔开, 相当于逻辑运算 “或”。实例代码如下:

```
@media (min-width: 700px), handheld { ... } /* 宽度不小于 700, 或者手持设备 */
```

采用 “not” 实现多个条件 “非” 的逻辑运算。实例代码如下:

```
@media not (min-width: 700px), handheld { ... }
```

注意: `not` 运算符是针对整体查询条件, 而非 “not” 后的第一个条件。

在上述实例代码中, 可以看到媒体查询的条件包含媒体类型和媒体属性。

常用的媒体类型有如下。

- `all`: 全部媒体类型。
- `print`: 打印设备 (常用于打印预览模式)。
- `screen`: 最主要的适用场景, 显示器。
- `speech`: 辅助设备。

注意: 在 CSS 2.1 和 Media Queries 3 中定义了一些额外的媒体类型, 如: `tty`、`tv`、`projection`、`handed`、`braille`、`embossed` 和 `aural`。这些在 Media Queries 4 中被弃用, 应当避免使用。

常用的媒体属性有如下。

- `width`: viewport 的宽度。
- `height`: viewport 的高度。
- `aspect-ratio`: viewport 的宽高比。
- `orientation`: 设备横竖屏, 值为 `portrait` 和 `landscape`。
- `resolution`: 设备分辨率 (像素密度), 可以采用每英寸 (dpi) 和每厘米 (dpcm) 表示。

在这些媒体属性中, 值为数值型可使用 “max” 和 “min” 前缀, 如 `max-width`、`min-height`。

“max”前缀表述的是查询条件不大于，“min”前缀表述的是查询条件不小于。

如在 15 个字符宽度或者更小的手持设备应用样式，代码如下：

```
@media handheld and (max-width: 15em) { ... }
```

采用媒体查询，可以给不同的设备应用不同的样式。这样在编写 CSS 的时候，先定义统一的样式，然后再根据不同的设备情况优化展示效果，比如根据设备的分辨率，给不同的设备加载不同尺寸的图片，从而使得在不同设备上都能得到更好的用户体验。

4.3.4 用 rem 开发响应式设计

CSS 中的计量单位包括 px、pt、em、rem 等，其中 em（font size of the element）是相对于当前元素的字体大小的计量单位，而 rem（font size of the root element）是指相对于根元素的字体的大小。虽然都是相对大小，但 em 由当前元素字号大小决定，无法做到统一管理，而 rem 则不同，不管用于几级元素，都直接由根元素的字号决定大小，很适合用来处理布局排布。

rem 单位主要用于移动 Web 开发，以适配不同尺寸的屏幕。下面看两个实际案例效果图，如图 4.21 和图 4.22 所示。

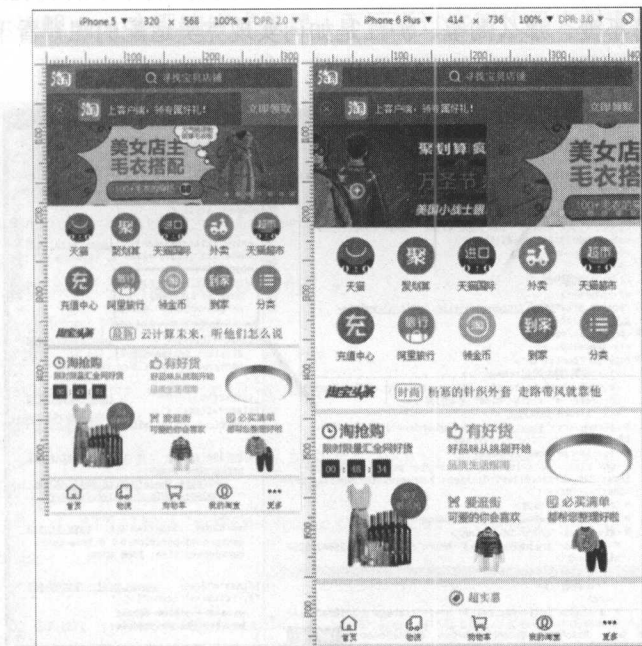


图 4.21 淘宝网在不同分辨率下呈现的效果



图 4.22 沪江网在不同分辨率下呈现的效果

通过图 4.21 和图 4.22 的两个案例，可以看出网页在不同分辨率下所呈现的效果几乎相同。整个页面的结构比例保持不变。那么淘宝、沪江是如何实现完美适配的呢？看下面的网页源码，答案即可揭晓，如图 4.23 和图 4.24 所示。

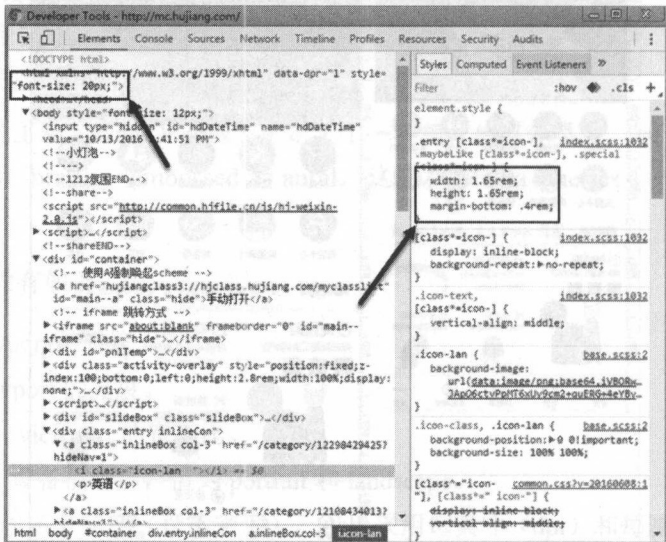


图 4.23 沪江网在 iPhone 5 下的网页源码

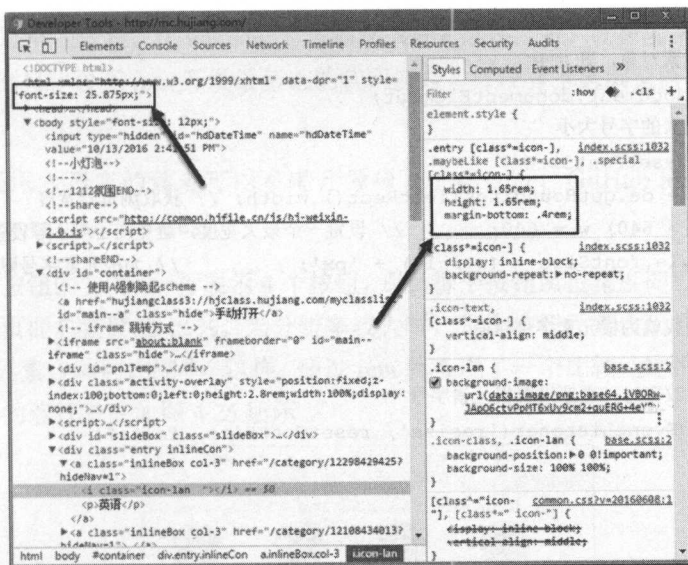


图 4.24 沪江网在 iPhone 6 Plus 下的网页源码

通过网页源码，发现页内元素是通过 rem 控制大小的，根据不同的分辨率对根元素设置不同的字号。为了更好地讲解 rem 实现响应式原理，下面通过实例代码来实现沪江头部的按钮布局，代码如下：

```

01 <html>
02 <head>
03 <!-- 设置viewport 为设备宽度 -->
04 <meta name="viewport" content="width=device-width" />
05 <style type="text/css">
06     /* 设置页面基础字号，并清除页面的默认边距 */
07     body { font-size:12px; margin:0; }
08     /* 设置容器宽度，并居中显示 */
09     .btns { width:10rem; margin:0 auto; }
10     /* 定义每个按钮的样式 */
11     .btns > a { float:left; width:2.5rem; text-align:center; padding-top:0.2rem; }
12     /* 定义按钮的子元素圆 */
13     .btns > a > i { display: inline-block; width: 1.2rem; height: 1.2rem; background:
gray; border-radius: 50%; }
14     /* 定义按钮文字样式 */
15     .btns > a > span { display:block; line-height:0.8rem; font-size:14px; }
16 </style>
17 <script>

```



```

19 (function () {
20     // 获取根元素对象, 既 HTML
21     var de = document.documentElement;
22     // 重置根元素的字号大小
23     function resetFontSize() {
24         var w = de.getBoundingClientRect().width; // 获取浏览器宽度
25         if (w > 640) w = 640; // 设置一个最大宽度, 避免在 iPad 等设备下过渡放大
26         de.style.fontSize = (w / 10) + 'px'; // 为根元素字号赋值
27     }
28     // 页面一加载就为根元素字号赋值
29     resetFontSize();
30     // 页面大小改变时, 重新设置根元素字号
31     window.addEventListener('resize', resetFontSize, false);
32 })();
33 </script>
34 </head>
35 <body>
36     <!-- 定义按钮容器和 8 个按钮元素 -->
37     <div class="btns">
38         <a><i></i><span>英语</span></a>
39         <a><i></i><span>日语</span></a>
40         <a><i></i><span>韩语</span></a>
41         <a><i></i><span>小语种</span></a>
42         <a><i></i><span>留学</span></a>
43         <a><i></i><span>职场</span></a>
44         <a><i></i><span>中小幼</span></a>
45         <a><i></i><span>全部分类</span></a>
46     </div>
47 </body>
48 </html>

```

从上面的代码中, 读者首先会看到一个 name 属性值为“viewport”的 META 标签声明, Viewport 是 HTML 5 引入的新特性, 用于控制网页视口。

提示: Viewport 更多的介绍请参考 http://www.w3schools.com/css/css_rwd_viewport.asp。

rem 单位是相对于网页根元素的字号大小而定, 所以实现 rem 布局开发时, 首先要做的就是对根元素的字号赋值, 参看代码第 18~33 行。那么问题来了, 到底赋值多少才合适呢? 这个并没有统一标准, 为了便于计算, 实例中使用实际宽度除以 10 的大小作为根元素字号。实际宽度除以 10 后, 相当于将页面进行 10 等分, 然后再对页面元素进行布局时, 就可以根据这个基数进行设置大小和距离。

另外考虑到横竖屏的变换,在 window 对象添加 `resize` 事件监听,以在屏幕变换后重置根元素字号。显然,本实例只包含了 `rem` 开发最基础的代码实现,并没有考虑浏览器兼容和 `DPR` 匹配问题。

提示:有兴趣深入研究的读者可以参考开源项目 `Flexiable`,GitHub 地址为 <https://github.com/amfe/lib-flexible>。

回到实现 8 个按钮的实例,一行展示 4 个按钮,那么每个按钮所占宽度即“ $10\text{rem}/4=2.5\text{rem}$ ”,相当于元素始终占页面总宽度的 25%。当分辨率较大时,每个按钮元素自动同比放大,当分辨率较小时,每个按钮元素自动同比缩小。这样,通过 `rem` 就实现了一个比较稳定可控的响应式布局。在 iPhone 6 Plus 下的运行效果如图 4.25 所示。

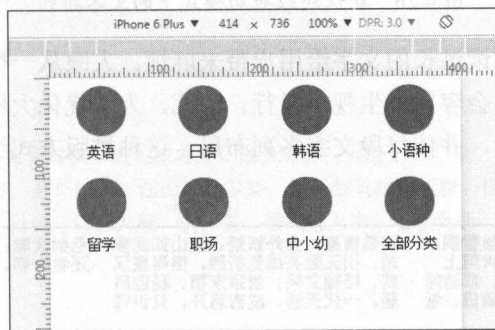


图 4.25 沪江头部按钮效果图

注意:并非所有单位都需要转换成 `rem`,比如字体大小,通常情况下仍然用 `px` 来设置,还有一些容器中的元素,直接使用百分比也是比较常用的做法。在实际开发中,应该以 `rem` 单位为基础,同时结合其他单位协同工作,这样才能开发出适配完美的网页。

4.3.5 多列 (Multiple Columns)

CSS 3 多列布局是块级布局模式的扩展,允许通过简单的定义实现文本的多列布局效果。默认在块级布局模式下,文本是从左至右排列,直至到达右边界,换一行再从左至右排列,代码如下:

```
<html>
<head>
</head>
<body>
```

```
<div>
    北国风光，千里冰封，万里雪飘。望长城内外，惟余莽莽；大河上下，顿失滔滔。山舞银蛇，原驰蜡象，欲与
    天公试比高。须晴日，看红装素裹，分外妖娆。江山如此多娇，引无数英雄竞折腰。惜秦皇汉武，略输文采；唐宗宋祖，
    稍逊风骚。一代天骄，成吉思汗，只识弯弓射大雕。俱往矣，数风流人物，还看今朝。
</div>
</body>
</html>
```

运行效果如图 4.26 所示。

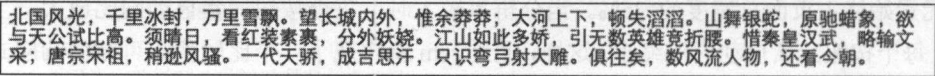


图 4.26 默认块级布局模式下的文本排列

这种类型的排列导致阅读过长的文字给用户带来麻烦。人眼从一行过长的文字末尾移到下一行开始处需要的时间较长，会容易产生视差错行。因此，为了优化大屏上的文字阅读体验，设计者应该限制文字段落的宽度，并使每段文字多列布局，这种排版方式经常出现在报刊中，效果如图 4.27 所示。

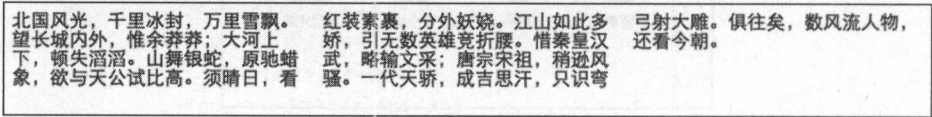


图 4.27 多列布局模式下的文本排列

用传统的方式实现文字的多列排版非常烦琐，但是通过 CSS 3 提供的多列布局就非常简单，代码如下：

```
<html>
<head>
<style>
#coll { column-count: 3; }          /* 文本分成三列布局 */
</style>
</head>
<body>
    <div id="col">
        北国风光，千里冰封，万里雪飘。望长城内外，惟余莽莽；大河上下，顿失滔滔。山舞银蛇，原驰蜡象，欲与
        天公试比高。须晴日，看红装素裹，分外妖娆。江山如此多娇，引无数英雄竞折腰。惜秦皇汉武，略输文采；唐宗宋祖，
        稍逊风骚。一代天骄，成吉思汗，只识弯弓射大雕。俱往矣，数风流人物，还看今朝。
    </div>
</body>
```

```
</html>
```

注意：就像本节开始的第一句话所说，多列布局是块级布局模式的扩展，这意味着多列布局必须应用在块级布局基础之上。比如上面的例子，如果把文本的父元素由 DIV 改成 SPAN，多列布局将不起作用。

上面的例子满足的需求是把文本固定分成多少列，而每列的宽度是不定的。如果需求是固定每列文本宽度，而具体有多少列不定，使用多列布局的实现代码如下：

```
<html>
<head>
<style>
#col1 { column-width: 200px; }      /* 固定每列文本宽度为 200px */
</style>
</head>
<body>
  <div id="col">
    北国风光，千里冰封，万里雪飘。望长城内外，惟余莽莽；大河上下，顿失滔滔。山舞银蛇，原驰蜡象，欲与天公试比高。须晴日，看红装素裹，分外妖娆。江山如此多娇，引无数英雄竞折腰。惜秦皇汉武，略输文采；唐宗宋祖，稍逊风骚。一代天骄，成吉思汗，只识弯弓射大雕。俱往矣，数风流人物，还看今朝。
  </div>
</body>
</html>
```

运行效果如图 4.28 所示。

北国风光，千里冰封，万里雪飘。望长城内外，惟余莽莽；大河上下，顿失滔滔。山舞银	蛇，原驰蜡象，欲与天公试比高。须晴日，看红装素裹，分外妖娆。江山如此多娇，引无	数英雄竞折腰。惜秦皇汉武，略输文采；唐宗宋祖，稍逊风骚。一代天骄，成吉思汗，只	识弯弓射大雕。俱往矣，数风流人物，还看今朝。
---	---	---	------------------------

图 4.28 固定每列宽度的多列布局

从图 4.28 可以看出，多列布局时，列与列之间存在间距。实际上，列与列之间不但有间距，还存在分隔标记，并且间距大小，以及分隔标记的大小、颜色和样式均可被自定义，代码如下：

```
<html>
<head>
<style>
#col1 {
  column-width: 200px;      /* 固定每列文本宽度为 200px */
  column-gap: 30px;        /* 列间距为 30px */
  column-rule: 10px solid red; /* 设置列标记的宽度，样式，颜色 */
}
</style>
```



```

</head>
<body>
  <div id="col">
    北国风光，千里冰封，万里雪飘。望长城内外，惟余莽莽；大河上下，顿失滔滔。山舞银蛇，原驰蜡象，欲与天公试比高。须晴日，看红装素裹，分外妖娆。江山如此多娇，引无数英雄竞折腰。惜秦皇汉武，略输文采；唐宗宋祖，稍逊风骚。一代天骄，成吉思汗，只识弯弓射大雕。俱往矣，数风流人物，还看今朝。
  </div>
</body>
</html>

```

运行效果如图 4.29 所示。

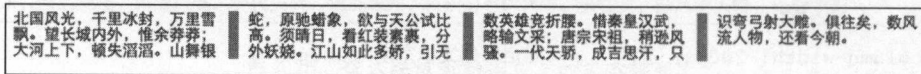


图 4.29 设置了列间距样式的多列布局

到此为止，多列布局的使用场景和使用方法已经介绍完毕。需要强调的一点是，多列布局是针对文本排版的布局，跟通常页面的左、中、右三列布局是两个概念，千万不要因为名字的原因而混淆。

4.4 动效

4.4.1 转换 (Transform)

W3C 组织分别在 2009 年 3 月发布了 3D 变形动画标准草案和 2D 变形动画标准草案，草案允许开发者对元素进行移动、缩放、旋转和倾斜。目前 CSS 3 Transform 属性已被各主流浏览器所支持。

提示：3D 变形动画标准草案，地址为 <http://www.w3.org/TR/CSS 3-3dtransforms/>。2D 变形动画标准草案，地址为 <http://www.w3.org/TR/CSS 3-2dtransforms/>。

Transform 语法如下：

```
transform : none | <transform-function> [<transform-function>]*;
```

<transform-function>指变形函数，可以是一个变形函数或多个变形函数的列表，具体支持的函数如下。

1. 移动 (Translate)

- **translate(x,y):** 定义 2D 位移转换，将元素从当前位置移动到给定的坐标点，参数 x、y 的可选单位有 px、pt、em、rem 等。参数 x 表示相对于原位置的 X 轴偏移距离，参数 y 表示

相对于原位置的 Y 轴偏移距离。参数 y 为可选参数，默认值为 0，表示 Y 轴偏移距离为 0 像素，此时 `translate` 等同于 `translateX`。

- `translate3d(x,y,z)`: 定义 3D 位移转换，使一个元素在三维空间内移动。其中参数 z 表示沿 Z 轴位移向量的长度。值越大表示距离观看者越近，元素也就越大，值越小（负值）表示距离观看者越远，元素也就越小。
- `translateX(x)`: 定义 2D 位移转换，只用 X 轴的值，等同于 `translate(x,0)`。
- `translateY(y)`: 定义 2D 位移转换，只用 Y 轴的值，等同于 `translate(0,y)`。
- `translateZ(z)`: 定义 3D 位移转换，只用 Z 轴的值，等同于 `translate3d(0,0,z)`。

2. 缩放 (Scale)

- `scale(x,y)`: 定义 2D 缩放转换，将元素按指定比例进行放大或缩小，参数 x 和 y 为 0 至 1 的浮点数。以元素的中心位置作为中心点，在水平 (X 轴) 和垂直 (Y 轴) 两个方向进行缩放。默认值为 1，大于 1 时元素放大，小于 1 时元素缩小。
- `scale3d(x,y,z)`: 定义 3D 缩放转换，使一个元素在三维空间内放大或缩小。其中参数 z 表示沿 Z 轴缩放的比例，默认值为 1，大于 1 时元素放大，小于 1 时元素缩小。值得注意的是单独使用 `scale3d` 函数不会看到什么效果，需要配合其他变形函数一起使用。
- `scaleX(x)`: 定义 2D 缩放转换，只用 X 轴的值，等同于 `scale(x,1)`。
- `scaleY(y)`: 定义 2D 缩放转换，只用 Y 轴的值，等同于 `scale(1,y)`。
- `scaleZ(z)`: 定义 3D 缩放转换，只用 Z 轴的值，等同于 `scale3d(1,1,z)`。

3. 旋转 (Rotate)

- `rotate(angle)`: 定义 2D 旋转，将元素按指定角度进行顺时针或逆时针的旋转变形，参数 `angle` 的单位有角度 (deg)、梯度 (grad)、弧度 (rad) 和圈 (turn)。默认以元素的中心位置为中心点，将元素在 2D 空间上旋转指定角度。如果参数设置为正数，表示顺时针旋转，如果参数设置为负数，表示逆时针旋转。
- `rotate3d(x,y,z,angle)`: 定义 3D 旋转，使一个元素在三维空间内旋转。参数 x 、 y 、 z 表示元素旋转的矢量值，取值范围为 0 到 1 的浮点数，参数 `angle` 表示元素旋转的角度。
- `rotateX(angle)`: 定义沿着 X 轴的 3D 旋转，等同于 `rotate3d(1,0,0,angle)`。
- `rotateY(angle)`: 定义沿着 Y 轴的 3D 旋转，等同于 `rotate3d(0,1,0,angle)`。
- `rotateZ(angle)`: 定义沿着 Z 轴的 3D 旋转，等同于 `rotate3d(0,0,1,angle)`。

4. 倾斜 (Skew)

- `skew(x-angle,y-angle)`: 定义 2D 倾斜转换（扭曲），将元素按指定角度进行倾斜变形。参

数 $x\text{-angle}$ 和 $y\text{-angle}$ 的单位有 deg 、 grad 、 rad 和 turn 。参数 $x\text{-angle}$ 定义元素在 X 轴上的倾斜角度, 参数 $y\text{-angle}$ 定义元素在 Y 轴上的倾斜角度。默认变形基点为元素的中心位置。

- `skewX(angle)`: 定义沿着 X 轴的 2D 倾斜转换, 等同于 `skew(x-angle, 0deg)`。
- `skewY(angle)`: 定义沿着 Y 轴的 2D 倾斜转换, 等同于 `skew(0deg, y-angle)`。

5. 其他

- `matrix(n,n,n,n,n,n)`: 定义 2D 自定义转换。使用 6 个值的 3 乘 3 矩阵的形式指定一个 2D 变换, 相当于直接应用一个 “[a b c d e f]” 变换矩阵, 即基于 X 轴和 Y 轴重新定位元素。本质上 Transform 中的所有 2D 变形效果都是由 `matrix` 实现的。
- `matrix3d(n,n,n,n,n,n,n,n,n,n,n,n,n,n,n,n)`: 定义 3D 自定义转换。使用 16 个值的 4 乘 4 矩阵, 使元素在三维空间内 (X 轴、 Y 轴和 Z 轴) 做指定的变形, 原理跟 `matrix` 的 2D 变形类似, 本质上 Transform 中的所有 3D 变形效果都是由 `matrix3d` 实现的。
- `perspective(n)`: 为 3D 转换元素定义透视视图。`perspective` 对于 3D 变形来说至关重要, 该属性用于设置查看者的位置, 并将可视内容映射到一个视锥上。如果不指定透视, 则 Z 轴空间相当于不存在。

Transform 的各种变形效果可以通过下面一张实例效果图来粗略体现, 如图 4.30 所示。

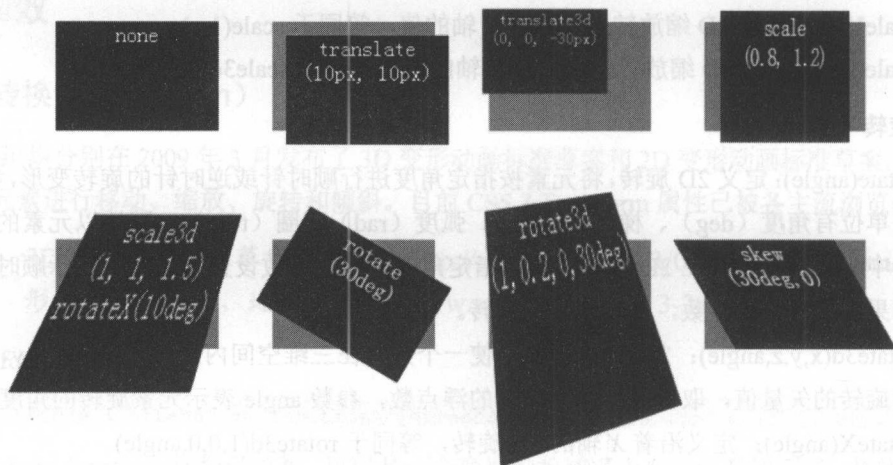


图 4.30 Transform 实例效果图

图 4.30 中效果分别使用了 `none`、`translate`、`translate3d`、`scale`、`scale3d`、`rotate`、`rotate3d` 和 `skew` 函数来定义元素 2D 和 3D 转换, 通过该效果图可以对这些转换函数有一个更清晰的认识, 代码如下:

```

01 <html>
02 <head>
03     <style type="text/css">
04         /* 定义一个容器, 并指定观察者距离为 100 像素 */
05         .box {width: 640px; margin: 0 auto; perspective: 100px;}
06         /* 定义容器中的每一项, 设置背景颜色为灰色 */
07         .box> div {float: left; width: 120px; height: 80px; margin: 40px 20px; 08
background-color: gray;}
09         /* 定义变形元素, 设置背景颜色为黑色 */
10         .box a {display: block; height: 80px; background-color: black; text-align: center;
11 color: white;}
12         /* transform 的默认值: none */
13         .box .none {transform: none;}
14         /* 2D 位移: 向右方和下方分别移动 10 像素 */
15         .box .translate {transform: translate(10px, 10px);}
16         /* 3D 位移: 沿 Z 轴向内位移 30 像素 */
17         .box .translate3d {transform: translate3d(0, 0, -30px);}
18         /* 2D 缩放: 沿 X 轴缩小到 0.8 倍, 沿 Y 轴放大到 1.2 倍 */
19         .box .scale {transform: scale(0.8, 1.2);}
20         /* 3D 缩放: X 轴和 Y 轴比例不变, 沿 Z 轴放大到 1.5 倍
21 (为便于观察变化, 同时将元素旋转 10 度) */
22         .box .scale3d {transform: scale3d(1, 1, 1.5) rotateX(10deg);}
23         /* 2D 旋转: 将元素顺时针旋转 30 度 */
24         .box .rotate {transform: rotate(30deg);}
25         /* 3D 旋转: 在 3D 空间 X=1, Y=0.2, Z=0 的向量上, 顺时针旋转 30 度 */
26         .box .rotate3d {transform: rotate3d(1, 0.2, 0, 30deg);}
27         /* 2D 倾斜: 将元素在 X 轴上倾斜 30 度 */
28         .box .skew {transform: skew(30deg, 0);}
29     </style>
30 </head>
31 <body>
32     <!-- 定义容器 -->
33     <div class="box">
34         <!-- 不变形 -->
35         <div><a class="none">none</a></div>
36         <!-- 2D 位移变形 -->
37         <div><a class="translate">translate<br />(10px, 10px)</a></div>
38         <!-- 3D 位移变形 -->
39         <div><a class="translate3d">translate3d<br />(0, 0, -30px)</a></div>
40         <!-- 2D 缩放变形 -->
41         <div><a class="scale">scale<br />(0.8, 1.2)</a></div>

```



```

42      <!-- 3D 位移变形 -->
43      <div><a class="scale3d">scale3d<br />(1, 1, 1.5)<br />rotateX(10deg)</a></div>
44      <!-- 2D 旋转变形 -->
45      <div><a class="rotate">rotate<br />(30deg)</a></div>
46      <!-- 3D 旋转变形 -->
47      <div><a class="rotate3d">rotate3d<br />(1,0.2,0,30deg)</a></div>
48      <!-- 2D 倾斜变形 -->
49      <div><a class="skew">skew<br />(30deg,0)</a></div>
50  </div>
51 </body>
52 </html>

```

注意: 3D 变换函数需要配合 perspective 使用, 变形基点 (中心点) 可以通过设置 transform-origin 属性来修改。有关 transform 的更多介绍, 请参考 http://www.w3schools.com/cssref/CSS3_pr_transform.asp。

4.4.2 过渡 (Transition)

CSS 3 动画包含 Transition 和 Animation 两种, 都可以通过改变元素的 CSS 属性来实现动画效果。本节内容主要介绍 CSS 3 Transition 的用法。有关 CSS 3 Transition 的详细信息可参阅 <http://www.w3.org/TR/CSS3-transitions/>。

Transition 语法如下:

```
transition : <property> <duration> <timing-function> <delay>;
```

CSS 3 Transition 适用于所有元素, 包括伪元素 “:before” 和 “:after”, 另外也支持设置多组动画, 语法是用半角逗号分隔每一组动画设置。下边针对 Transition 的每个属性逐一介绍。

1. transition-property

规定应用过渡效果的 CSS 属性的名称, 当元素的该 CSS 属性发生改变时, 过渡效果开始执行 (从旧的属性值过渡到新的属性值), 语法如下:

```
transition-property : none | all | <property>;
```

其中 all 表示设置所有的 CSS 属性。

2. transition-duration

规定完成过渡效果需要花费的时间。属性值为浮点数, 单位可以秒 (s) 或毫秒 (ms) 计, 语法如下:

```
transition-duration : <time>;
```

如果过渡时间设置为 0，则没有过渡效果，直接看到结果。

3. transition-timing-function

规定过渡效果的速度曲线。该属性允许过渡效果随着时间来改变其速度。语法如下：

```
transition-timing-function : linear | ease | ease-in | ease-out | ease-in-out |  
cubic-bezier(n,n,n,n)
```

- **linear**: 规定以相同的速度从开始到结束的过渡效果。等同于贝济埃曲线(0.0, 0.0, 1.0, 1.0)。
- **ease**: 规定逐渐变慢的过渡效果。等同于贝济埃曲线(0.25, 0.1, 0.25, 1.0)。
- **ease-in**: 规定慢速开始的过渡效果。等同于贝济埃曲线(0.42, 0, 1.0, 1.0)。
- **ease-out**: 规定慢速结束的过渡效果。等同于贝济埃曲线(0, 0, 0.58, 1.0)。
- **ease-in-out**: 规定慢速开始和结束的过渡效果。等同于贝济埃曲线(0.42, 0, 0.58, 1.0)。
- **cubic-bezier(n,n,n,n)**: 允许自定义一个时间曲线。通过自定义的贝济埃曲线来计算过渡过程中的属性值。

4. transition-delay

规定过渡效果开始的延迟时间。属性值为浮点数，单位可以秒(s)或毫秒(ms)计。默认时间为 0，即指过渡立即执行，没有延迟。语法如下：

```
transition-delay : <time>
```

需要注意的是，**transition-delay** 和 **transition-duration** 都是设置时间，所以在连写中要严格区分所在位置，一般浏览器会把第一个时间值解析为 **duration**，第二个时间值解析为 **delay**。

通过下面一个模拟树叶落下的实例，帮助读者学习 CSS 3 Transition 的用法，代码如下：

```
01 <html>  
02 <head>  
03 <style type="text/css">  
04 .img { position:fixed; top:-400px; } /* 设置图片初始位置 */  
05 .img { transition:top ease 1s; } /* 设置图片过渡效果 */  
06 .img.show { top:20px; } /* 设置图片展示后的位置 */  
07 </style>  
08 </head>  
09 <!-- 在页面加载后对 img 应用新的样式 -->  
10 <body onload="document.querySelector('.img').className+=' show'">  
11 <!-- 定义一张默认隐藏的图片 -->  
12 
```

```
13 </body>
14 </html>
```

实例中树叶默认隐藏在头部，当页面加载完成的时候，树叶以逐渐变慢的过渡效果向下掉落，直至停留在页面中间。如图 4.31 所示，页面加载完成，树叶开始向箭头方向下落。

根据 CSS 中的设置，树叶下落过程持续 1 秒钟，然后停留在页面中间保持静止状态。如图 4.32 所示，页面已经完全落下。

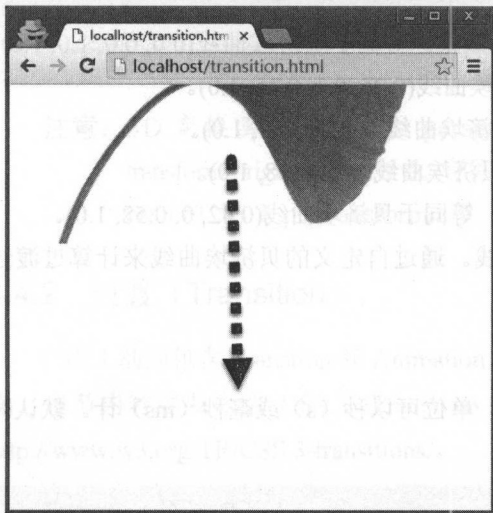


图 4.31 Transition 过渡效果进行中

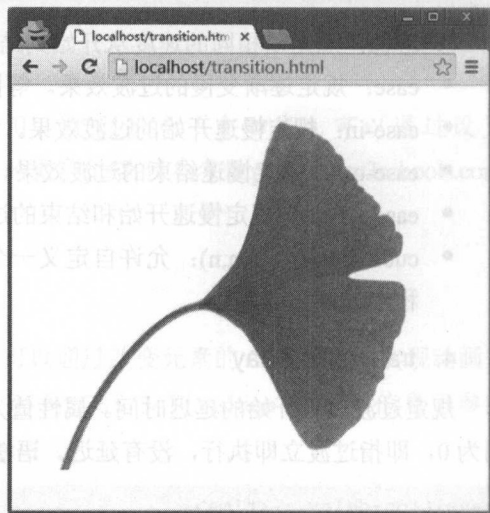


图 4.32 Transition 过渡效果完成

注意：实例中省略了 Transition 的第 4 个属性值 `transition-delay`，省略时表示取默认值 0，即不延迟，立即执行过渡。另外 Transition 的 4 个属性值也可以单独来定义。有关 Transition 的更多介绍，请参考 http://www.w3schools.com/cssref/CSS3_pr_transition.asp。

4.4.3 动画 (Animation)

CSS 3 动画包含 Transition 和 Animation 两种，在上一节介绍了 Transition 的用法。本节内容将介绍 CSS 3 动画的第 2 种高级用法 Animation。有关 CSS 3 Animation 的详细信息请参阅 <http://www.w3.org/TR/CSS3-animations/>。

Animation 语法如下：

```
animation : <name> <duration> <timing-function> <delay> <iteration-count> <direction>
```

CSS 3 Animation 适用于所有块状元素和内联元素，通过定义动画中的关键帧来实现复杂的动

画效果。Animation 属性的初始值根据其各个子属性的默认值而定。下边针对 Animation 的每个属性逐一介绍。

1. animation-name

定义动画名称，语法如下：

```
animation-name : none | IDENT;
```

animation-name 的属性值为“@keyframes”创建的动画名称，默认值为 none。如果动画名称为 none，则不会有任何动画效果。

2. animation-duration

定义动画播放的时间，语法如下：

```
animation-duration : <time>;
```

animation-duration 的属性值为浮点数，单位可以秒（s）或毫秒（ms）计。初始值为 0，表示没有动画效果，直接过渡到最后一帧。该属性的使用方法跟 transition-duration 的使用方法一致。

3. animation-timing-function

定义动画播放方式，语法如下：

```
animation-timing-function : linear | ease | ease-in | ease-out | ease-in-out |  
cubic-bezier(n,n,n,n);
```

animation-timing-function 属性的默认值是 ease，其使用方法和 transition-timing-function 相同。对于每一种变换方式的介绍，可以参考 4.4.2 节。

4. animation-delay

定义动画延迟播放的时间，语法如下：

```
animation-delay : <time>;
```

animation-delay 的属性值为浮点数，单位可以秒（s）或毫秒（ms）计。初始值为 0，表示动画立即执行，如果定义了时间，则表示动画在定义的时间后才开始执行。该属性的用法同 transition-delay。

5. animation-iteration-count

定义动画播放的次数，语法如下：

```
animation-iteration-count : infinite | <number>;
```


`animation-iteration-count` 属性的默认值为 1, 表示动画只播放 1 次。值为 `infinite` 表示播放无限次, 也就是动画不停地循环播放, 永不停止。如果取值为非整数, 则动画将只播放周期中的一部分。

6. animation-direction

定义动画播放的方向, 语法如下:

```
animation-direction : normal | alternate;
```

`animation-direction` 属性的默认值为 `normal`, 指动画始终向前播放。如果赋值 `alternate`, 表示动画在第偶数次向前播放, 第奇数次反向播放。

在上一节的实例中演示了树叶下落的过渡效果, 本实例将在此基础上进行功能扩展, 使元素在过渡中含有多种变换效果, 代码如下:

```
01 <html>
02 <head>
03 <style type="text/css">
04 .img { position:fixed; top:-400px; width:500px;}          /* 设置容器样式 */
05 .img.show { animation:move 2s ease 1 normal;}            /* 设置图片动画效果 */
06 @keyframes move {                                       /* 设置动画效果中的关键帧 */
07     0% { top:-400px; }                                  /* 定义动画的初始状态 */
08     50% { top:20px; }                                    /* 下落到页面中间 */
09     100% { width:1000px; }                              /* 将图片放大 */
10 }
11 </style>
12 </head>
13 <!-- 在页面加载后对 img 应用新的样式 -->
14 <body onload="document.querySelector('.img').className+=' show'">
15     <!-- 定义一张默认隐藏的图片 -->
16     
17 </body>
18 </html>
```

实例中树叶默认隐藏在头部, 当页面加载完成的时候, 树叶以逐渐变慢的过渡效果向下掉落, 停留在页面中间, 然后再慢慢放大。如图 4.33 所示, 页面加载完成, 树叶开始下落, 并在动画时间轴的 50% 处停留在页面中间。

树叶落下之后动画开始执行第二种过渡效果, 树叶被逐渐放大, 最终宽度至 1000 像素, 如图 4.34 所示。

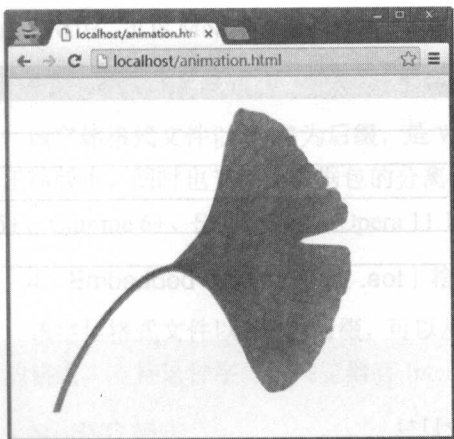


图 4.33 Animation 动画运行到 50% 的效果图

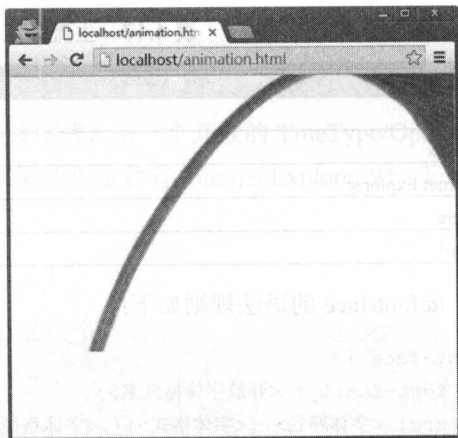


图 4.34 Animation 动画运行到 100% 的效果图

注意：CSS 3 Animation 跟 Transition 不同的是，Animation 只是将定义好的动画应用到元素 1 次或 n 次，并不会对元素最初的样式造成影响，也就是说，在“@keyframes”中定义的 CSS 属性，最终都会消失。有关 Animation 的更多介绍，请参考 http://www.w3schools.com/cssref/CSS3_pr_animation.asp。

4.5 常用特性

本节将介绍 CSS 3 中一些常用特性，包括开放字体格式、背景、颜色、文字效果、边框和用户界面。这些特性是组成页面的重要部分，只有真正理解才能更好地完成移动端 Web 开发。

4.5.1 开放字体格式（WOFF）

开放字体格式（Web Open Font Format，简称 WOFF），是一种网页采用的字体格式标准。WOFF 包含了基于 SFNT 的字体（如 PostScript、TrueType 和 OpenType 等开放字体格式）。这种字体格式的优点是能有效利用压缩减少档案大小，并且不包含加密也不受 DRM（数位著作权管理）限制。

可以通过 CSS 3 的 @font-face 来使用 WOFF，虽然 @font-face 是 CSS 3 中的一个模块，但该功能早在 Internet Explorer 4 中就被支持。表 4.17 列举了目前主流浏览器对 @font-face 的支持情况。

表 4.17 主流浏览器对“@font-face”的支持情况

浏 览 器	支 持 版 本
Chrome	4.0+
Firefox	3.5+
Internet Explorer	6+
Opera	10.1+
Safari	3.2+

@font-face 的语法规则如下:

```
@font-face {
  font-family: <开放字体格式名>;
  src: <字体路径> [<字体格式>][,<字体路径> [<字体格式>]]*;
  [font-weight: <是否粗体>];
  [font-style: <字体样式>];
}
```

语法规则说明如下。

- font-family: 取值为自定义的字体名字, 如“font-family: someFontName”。
- src: 字体路径是自定义字体的存放路径, 可以是相对路径也可以是绝对路径。字体格式是自定义的字体格式, 主要是方便浏览器识别, 其值主要类型有 truetype、opentype、truetype-aat、embedded-opentype、avg 等。
- font-weight: 默认值为 normal, 其他取值如 bold、100~900。
- font-style: 默认值为 normal, 其他取值如 italic、oblique 等。

注意: 虽然 Internet Explorer 4+就开始支持@font-face, 但这里涉及一个字体 format 的问题, 不同的浏览器对字体格式的支持是不一致的。

下面针对不同字体格式做一个说明。

1. TureType (.ttf) 格式

该字体格式文件以.ttf 为后缀, 是操作系统 Windows 和 Mac 上比较常见的字体, 支持这种字体的浏览器有 Internet Explorer 9+、Firefox 3.5+、Chrome 4+、Safari 3+、Opera 10+、iOS Mobile Safari 4.2+。

2. OpenType (.otf) 格式

该字体格式文件以.otf 为后缀, 被认为是一种原始的字体格式, 内置在 TureType 基础上, 提供了比 TrueType 格式更多的功能, 支持这种字体的浏览器有 Firefox 3.5+、Chrome 4.0+、Safari 3.1+、

Opera 10.0+、iOS Mobile Safari 4.2+。

3. Web Open Font Format (.woff) 格式

该字体格式文件以.woff为后缀,是Web字体中最佳格式,是一个开放的TrueType/OpenType的压缩版本,同时也支持元数据包的分离,支持这种字体的浏览器有Internet Explorer 9+、Firefox 3.5+、Chrome 6+、Safari 3.6+、Opera 11.1+。

4. Embedded Open Type (.eot) 格式

该字体格式文件以.eot为后缀,可以从TrueType创建此格式的字体,是Internet Explorer的专用的格式,支持这种字体的浏览器有Internet Explorer 4+。

5. SVG 格式

该字体是基于SVG字体渲染的一种格式,支持SVG字体的浏览器有Chrome 4+、Safari 3.1+、Opera 10.0+、iOS Mobile Safari 3.2+。

在移动Web开发的项目中往往并不需要兼容Internet Explorer浏览器,但是其他浏览器的兼容问题仍然存在,单使用上述的任何一种字体格式文件都无法全部适配。为了适配更多的浏览器,使用语法格式如下:

```
@font-face {
    font-family: 字体名;
    src: url('字体名.eot'); /* IE9 Compat Modes */
    src: url('字体名.eot?#iefix') format('embedded-opentype'), /* IE6-IE8 */
    url('字体名.woff') format('woff'), /* Modern Browsers */
    url('字体名.ttf') format('truetype'), /* Safari, Android, iOS */
    url('字体名.svg#字体名') format('svg'); /* Legacy iOS */
}
```

下面通过一个实际的例子来演示@font-face的使用。首先前往<http://fontello.com>下载fontello字体,并存放在本地的fonts文件夹中,代码如下:

```
01 <!DOCTYPE html>
02 <html>
03 <head>
04     <style>
05         @font-face {
06             font-family: 'fontello';
07             src: url('../font/fontello.eot?69798120');
08             src: url('../font/fontello.eot?69798120#iefix') format('embedded-opentype'),
09             url('../font/fontello.woff?69798120') format('woff'),
```



```

10         url('../font/fontello.ttf?69798120') format('truetype'),
11         url('../font/fontello.svg?69798120#fontello') format('svg');
12         font-weight: normal;
13         font-style: normal;
14     }
15     demo-icon {
16         font-family: "fontello";           /* 字体名 */
17         font-style: normal;                 /* 字体样式 */
18         display: inline-block;              /* 行内块元素 */
19         font-size: 42px;                    /* 字体大小 */
20         margin-right: 5px;                  /* 元素的右外边距 */
21     }
22 </style>
23 </head>
24 <body>
25     <i class="demo-icon icon-glass">&#xe800;</i>    /* 酒杯图标 */
26     <i class="demo-icon icon-music">&#xe801;</i>      /* 音乐图标 */
27     <i class="demo-icon icon-search">&#xe802;</i>      /* 搜索图标 */
28     <i class="demo-icon icon-mail">&#xe803;</i>        /* 信封图标 */
29 </body>
30 </html>

```

使用 Chrome 浏览器打开 font-woff.html 实例文件, 字体效果如图 4.35 所示。



图 4.35 字体效果

提示: 可以前往 <http://www.dafont.com> 或者 <https://fonts.google.com> 上下载自己喜欢的字体文件, 放在项目的字体文件夹中。 <https://www.fontsquirrel.com/tools/webfont-generator> 网站可以对字体文件进行转换, 并导出各种版本的字体格式。

4.5.2 背景 (Backgrounds)

background 属性用来设置元素的背景, 也是所有背景属性的简写方式, 也可以单独设置, 背景属性如下。

- **background-color:** 规定要使用的背景颜色。
- **background-position:** 规定背景图像的位置。

- **background-size**: 规定背景图片的尺寸。
- **background-repeat**: 规定如何重复背景图像。
- **background-origin**: 规定背景图片的定位区域。
- **background-clip**: 规定背景的绘制区域。
- **background-attachment**: 规定背景图像是否固定或者随着页面的其余部分滚动。
- **background-image**: 规定要使用的背景图像。

其中 **background-size**、**background-origin** 和 **background-clip** 是 CSS 3 中新增的属性，三种属性的主流浏览器兼容性见表 4.18。

表 4.18 主流浏览器对 CSS 3 新增背景属性的支持情况

Chrome	Firefox	IE	Opera	Safari
all	4+	9+	all	5+

background 的语法规则如下：

```
background: #000 url("图片地址") no-repeat left top;
```

取值说明如下。

- **#000**: 设置纯色背景。
- **url**: 设置背景图片的地址，可以是相对地址也可以是绝对地址。
- **no-repeat**: 设置图片平铺的方式，此处为“不重复”。
- **left**: 图片在容器中靠左显示。
- **top**: 图片在容器中靠上显示。

这种语法规则去设置图片背景比较常用，语法比较简洁，满足日常大部分的需求。同时，还支持单独设置每一项，代码如下：

```
background-color: #000;
background-image: url("图片地址");
background-repeat: no-repeat;
background-position: left top;
```

下面使用 **background** 设置背景图片，代码如下：

```
01 <html>
02 <head>
03 <style type="text/css">
04     .school-opens {
05         width: 250px;           /* 元素的宽度 */
```

```

06      height: 150px;           /* 元素的高度 */
07      border: 1px solid #000;  /* 元素的边框 */
08      /* url 指定了图片的路径。两个 center 是指左右和上下居中。
09      no-repeat 是指背景图像将仅显示一次，不重复。 */
10      background: url('../school_opens.png') center center no-repeat;
11  }
12 </style>
13 </head>
14   <body>
15     <div class="school-opens"></div>
16   </body>
17 </html>

```

使用 Chrome 浏览器打开 background.html 实例文件，背景效果如图 4.36 所示。



图 4.36 背景效果

可以看出，整个容器要比背景图片大出许多。通过设置背景属性值为“center center”之后，图片在容器中上下和左右居中。这个效果常常用来分离纯色背景和复杂图案，纯色背景用 background-color 实现，复杂图案用 background-image 实现，以此来减少图片的大小。

在 CSS 3 中，background 新增三个属性如下。

1. background-origin

background-origin 属性规定背景图片的定位区域，语法规则如下：

```
background-origin: padding-box | border-box | content-box
```

为了兼容新老版本的浏览器，在使用 background-origin 改变 background-position 的原点位置时，配合老语法一起使用，并且新语法放在老语法之后，这样只要支持新语法规则的浏览器会自动识别 background-origin，代码如下：

```

background-origin: padding | border | content;
background-origin: padding-box | border-box | content-box;

```

取值说明如下。

- padding-box(padding): 默认值, 决定 background-position 起始位置从 padding 的外边缘 (border 的内边缘) 开始显示背景图片。
- border-box(border): 此值决定 background-position 起始位置从 border 的外边缘开始显示背景图片。
- content-box(content): 此值决定 background-position 起始位置从 content 的外边缘 (padding 的内边缘) 开始显示背景图片。

通过 background-origin 实例比较三种取值, 使用 Chrome 浏览器打开 background-origin.html 文件, 效果如图 4.37 所示。

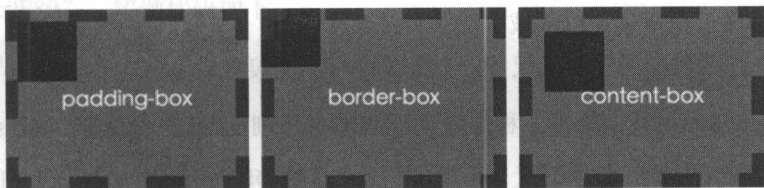


图 4.37 background-origin 效果

2. background-clip

background-clip 规定背景的绘制区域, 语法规则如下:

```
background-clip: border-box | padding-box | content-box;
```

取值说明如下。

- border-box: 默认值, 背景从 border 区域向外裁剪, 也就是超出部分将被裁剪。
- padding-box: 背景从 padding 区域向外裁剪, 超过 padding 区域的背景将被裁剪。
- context-box: 背景从 content 区域向外裁剪, 超过 context 区域的背景将被裁剪。

通过 background-clip 实例比较三种取值, 使用 Chrome 浏览器打开 background-clip.html 文件, 效果如图 4.38 所示。

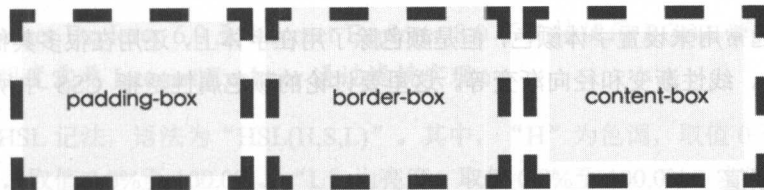


图 4.38 background-clip 效果

3. background-size

background-size 属性规定背景图片的尺寸,语法规则如下:

```
background-size: length | percentage | cover | contain;
```

取值说明如下。

- **length**: 设置背景图像的高度和宽度。第一个值设置宽度,第二个值设置高度。如果只设置一个值,则第二个值默认设置为“auto”。
- **percentage**: 以父元素的百分比来设置背景图像的宽度和高度。第一个值设置宽度,第二个值设置高度。如果只设置一个值,则第二个值默认设置为“auto”。
- **cover**: 把背景图像扩展至足够大,以使背景图像完全覆盖背景区域。背景图像的某些部分也许无法显示在背景定位区域中。
- **contain**: 把图像扩展至最大尺寸,以使其宽度和高度完全适应内容区域。

通过 background-size 实例比较三种属性,使用 Chrome 浏览器打开 background-size.html 文件,效果如图 4.39 所示。

CSS 3 还支持多重背景,可以把不同背景图片放到同一个元素中,多个图片 URL 之间使用逗号分隔。使用 Chrome 浏览器打开 background-multiple.html 文件查看多重背景,效果如图 4.40 所示。

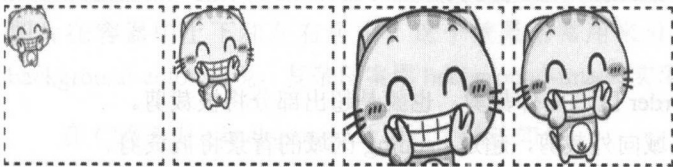


图 4.39 background-size 效果

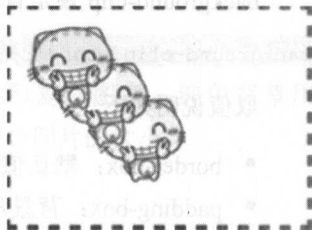


图 4.40 多重背景实例

4.5.3 颜色 (Color)

Color 属性通常用来设置字体颜色,但是颜色除了用在字体上,还用在很多其他的地方,比如元素的背景颜色、线性渐变和径向渐变等。这里要讨论的颜色属性泛指 CSS 中所有用到的颜色 Color。

1. 颜色的取值

Color 数据类型包括 Color Name、HEX、RGB、RGBA、HSL、HSLA、Transparent、CurrentColor。

下面介绍不同数据类型的使用。

- **Color Name (颜色名称)**：用颜色关键字来指定颜色，颜色关键字共包含 147 种。实例代码如下：

```
color: black;           /* 黑色字体 */
border-color: black;    /* 黑色边框 */
background: black;      /* 黑色背景 */
```

- **HEX**：十六进制记法表示颜色值，如“#RRGGBB”或“#RGB”。“RR”为红色值，“GG”为绿色值，“BB”为蓝色值，取值均为十六进制正整数，取值范围为 00 至 FF。如果三个参数各自两位数字相同可以缩写为“#RGB”格式。比如“color:#FF8800”，可以缩写为“color:#F80”。实例代码如下：

```
color: #000000;         /* 黑色字体 */
border-color: #000000;  /* 黑色边框 */
background: #000000;    /* 黑色背景 */
```

- **RGB**：RGB 记法，语法为“RGB(R,G,B)”，“R”为红色值，“G”为绿色值，“B”为蓝色值，取值为正整数或者百分数。正整数值的取值范围为 0 至 255。百分数值的取值范围为 0.0%至 100.0%。

正整数 255 对应百分比数值 100%，如下代码：

```
rgb(255,255,255) = rgb(100%,100%,100%) = #FFFFFF = #FFF。
```

RGB 实例代码如下：

```
color: rgba(0, 0, 0);    /* 黑色字体*/
border-color: rgba(0, 0, 0); /* 黑色边框 */
background: rgba(0, 0, 0); /* 黑色背景 */
```

- **RGBA**：色彩模式与 RGB 相同，在 RGB 模式上新增了 Alpha 透明度。实例代码如下：

```
color: rgba(0, 0, 0, 0.5); /* 黑色字体，50%的不透明度 */
border-color: rgba(0, 0, 0, 0.5); /* 黑色边框，50%的不透明度 */
background: rgba(0, 0, 0, 0.5); /* 黑色背景，50%的不透明度 */
```

提示：Internet Explorer 6.0 至 Internet Explorer 8.0 不支持使用 RGBA 模式实现透明度，透明度需要 Internet Explorer 通过滤镜实现。

- **HSL**：HSL 记法，语法为“HSL(H,S,L)”。其中，“H”为色调，取值 0 至 360。“S”为饱和度，取值 0.0%至 100.0%。“L”为亮度，取值 0.0%至 100.0%。实例代码如下：

```
color: hsl(360, 50%, 50%); /* 红色字体 */
```

```
border-color: hsl(360, 50%, 50%); /* 红色边框 */
background: hsl(360, 50%, 50%); /* 红色背景 */
```

- **HSLA**: 此色彩模式与 HSL 相同, 只是在 HSL 模式上新增了 Alpha 透明度。实例代码如下:

```
color: hsla(360, 50%, 50%, 0.5); /* 红色字体, 50%不透明度 */
border-color: hsla(360, 50%, 50%, 0.5); /* 红色边框, 50%不透明度 */
background: hsla(360, 50%, 50%, 0.5); /* 红色背景, 50%不透明度 */
```

- **Transparent**: 用来指定全透明色彩。Transparent 是全透明黑色 (black) 的速记法, 即一个类似 “rgba(0,0,0,0)” 这样的值。在 CSS1 中, Transparent 被用来作为 background-color 的一个参数值, 用于表示背景透明。在 CSS2 中, border-color 也开始接受 Transparent 作为参数值。在 CSS 3 中, Transparent 被延伸到任何一个有 Color 值的属性上。实例代码如下:

```
color: transparent; /* 字体颜色透明 */
border: 1px solid transparent; /* 边框颜色透明 */
background: transparent; /* 背景颜色透明 */
```

- **CurrentColor**: CSS 3 中标准的关键字, 表示当前颜色, 准确点说是当前的标签所继承的文字颜色。实例代码如下:

```
color: red; /* 文字颜色为红色 */
border: 1px solid currentcolor; /* 一像素的红色边框, 此处的 currentcolor 就是 red */
```

2. CSS 3 渐变 (Gradient)

CSS 3 Gradient 分为 linear-gradient (线性渐变) 和 radial-gradient (径向渐变)。

线性渐变的语法如下所示:

```
background: linear-gradient(direction, color-stop1, color-stop2, ...); /* 标准语法 */
background: -moz-linear-gradient(direction, color-stop1, color-stop2, ...); /* Firefox */
background: -webkit-linear-gradient(direction, color-stop1, color-stop2, ...); /* Chrome */
background: -o-linear-gradient(direction, color-stop1, color-stop2, ...); /* Opera */
```

参数 direction 为渐变方向, 默认是从上到下, 也可以设置具体的角度值, 其他参数为颜色值。

径向渐变渐变语法如下所示:

```
background: radial-gradient(position, shape size, start-color, ..., last-color); /* 标准语法 */
background: -moz-radial-gradient(position, shape size, start-color, ..., last-color); /* Firefox */
background: -webkit-radial-gradient(position, shape size, start-color, ..., last-color); /* Chrome */
background: -o-radial-gradient(position, shape size, start-color, ..., last-color); /* Opera */
```

参数 `position` 为渐变的起始位置, “`shape size`” 表示渐变的形状和大小, 其他参数为颜色值。

用 Chrome 打开 `gradient.html` 实例文件, 使用渐变设置背景颜色的效果如图 4.41 所示。

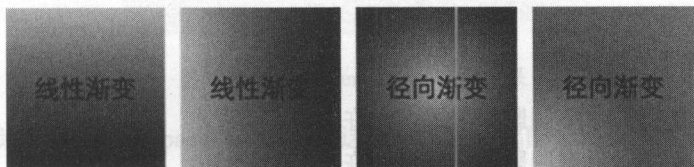


图 4.41 渐变背景效果

注意: 大面积的背景图片非常消耗流量, 这在移动开发中需要小心。所以, 当网页有大面积背景图片时, 可以和设计师商量, 设计适合用 CSS 制作的渐变图片, 然后交由前端开发用 CSS 代码实现, 节省传输带宽, 在性能和效果中寻找一个平衡点。

3. opacity

`opacity` 属性设置元素的不透明级别, 语法如下所示:

```
opacity: value|inherit;
```

`opacity` 属性会影响后代元素, 使后代元素也有相应的不透明度。所以, 如果元素中存在后代元素, 建议使用 `RGBA` 替代实现。用 Chrome 打开 `opacity.html` 实例文件, 分别用 `opacity` 和 `RGBA` 来设置不透明度, 效果如图 4.42 所示。

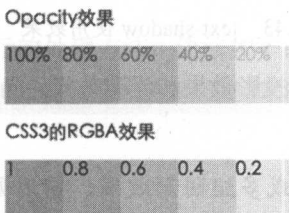


图 4.42 `opacity` 和 `RGBA` 来设置不透明度的效果

注意: `opacity` 常常用于改变大型单一元素的不透明度, 比如弹层的遮罩层等。而当元素的后代元素较多或结构较复杂时, 则应该选用 `RGBA` 来替代实现, 比如, 有半透明背景图片的大块文字区域等。

4.5.4 文字效果 (Text Effects)

CSS 3 对文字添加了多种效果, 下面介绍其中常用的 4 种。

1. text-shadow

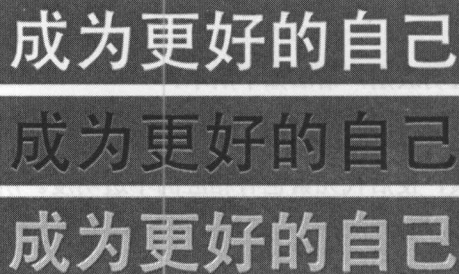
text-shadow 属性向文本设置阴影。语法如下所示:

```
text-shadow: h-shadow v-shadow blur color;
```

取值说明如下。

- h-shadow: 必需, 水平阴影的位置, 允许负值。
- v-shadow: 必需, 垂直阴影的位置, 允许负值。
- blur: 可选, 模糊的距离。
- color: 可选, 阴影的颜色。

使用 Chrome 浏览器打开 text-shadow.html 实例文件查看 text-shadow 使用效果, 如图 4.43 所示。



成为更好的自己
成为更好的自己
成为更好的自己

图 4.43 text-shadow 使用效果

实例中三个效果分别为辉光效果、投影效果和浮雕效果。当然 text-shadow 还可以创作更多的文字效果, 大家可以自行尝试。

注意: 可以给一个对象应用一组或多组阴影效果, 使用见文中语法, 用逗号隔开。

2. text-overflow

text-overflow 属性规定当文本溢出包含元素时发生的事情。语法如下所示:

```
text-overflow: clip | ellipsis | string;
```

取值说明如下。

- clip: 修剪文本。
- ellipsis: 显示省略符号来代表被修剪的文本。
- string: 使用给定的字符串来代表被修剪的文本。

在实际开发中，常常需要实现文本在一行内显示，并且文本过长时使用省略号替代。使用 Chrome 浏览器打开 text-overflow.html 文件，效果如图 4.44 所示。

公告：这是一个很长的通知，这是一...

图 4.44 text-overflow 省略号效果

text-overflow 通常和 white-space、overflow 这两个属性一起配合使用，实例代码如下：

```
white-space: nowrap;          /* 段落中的文本不换行 */
overflow: hidden;            /* 内容被修剪，超出元素框的内容不显示 */
text-overflow: ellipsis;     /* 显示省略符号来代表被修剪的文本 */
```

3. word-wrap

word-wrap 属性允许长单词或 URL 地址自动换行。语法如下：

```
word-wrap: normal | break-word;
```

取值说明如下。

- normal：只在允许的断字点换行（浏览器保持默认处理）。
- break-word：在长单词或 URL 地址内部进行换行。

使用 Chrome 浏览器打开 word-wrap.html 实例文件，效果如图 4.45 所示。

4. word-break

word-break 属性规定自动换行的处理方法。语法如下：

```
word-break: normal | break-all | keep-all;
```

取值说明如下。

- normal：使用浏览器默认的换行规则。
- break-all：允许在单词内换行。
- keep-all：只能在半角空格或连字符处换行。

使用 Chrome 浏览器打开 word-break.html 实例文件，效果如图 4.46 所示。

这是一个非常长的单词：
thisisaveryveryveryvery
veryverylongword. 这
个长单词会换行。

This is a
veryveryveryveryveryveryveryvery
long paragraph.

This is a veryveryveryve
ryveryveryveryver
yvery long paragraph.

图 4.45 word-wrap 使用效果

4.46 word-break 使用效果

4.5.5 边框 (Border)

Border 主要被用于添加元素的边框效果, 在日常的开发中使用非常普遍, 不仅能让网页看起来井井有条, 而且能让网页更加美观。CSS 3 为 Border 属性添加了更多的功能, 本节将介绍 Border 属性的用法和新增的几个功能。

1. Border 属性

首先来看下 Border 属性的基本语法:

```
border: border-width border-style border-color;
```

取值说明如下。

- border-width: 规定边框的宽度。
- border-style: 规定边框的样式。
- border-color: 规定边框的颜色。

可以按顺序单独设置这三个属性, 实例代码如下:

```
border-width: 1px;           /* 边框宽度为 1 像素 */
border-style: solid;        /* 边框的样式为实线 */
border-color: #000;         /* 边框的颜色为黑色 */
```

2. border-radius

border-radius 用于设置或检索对象使用圆角边框, 目前主流浏览器都能较好地支持, 在移动开发中出镜率也非常高, 常常用于制作圆角按钮、圆形图标、圆角内容区域等。

border-radius 是一个简写属性, 用于设置四个 “border-*-radius” 属性。border-radius 的语法如下所示:

```
border-radius: none | {1-4} length | % / {1-4} length | %;
```

注意: 按此顺序设置每个 radii 的四个值。如果省略 bottom-left, 则与 top-right 相同; 如果省略 bottom-right, 则与 top-left 相同; 如果省略 top-right, 则与 top-left 相同。

使用 Chrome 浏览器打开 border-radius.html 实例文件, 这是一个典型的用法, border-radius 设置圆角按钮的效果, 如图 4.47 所示。

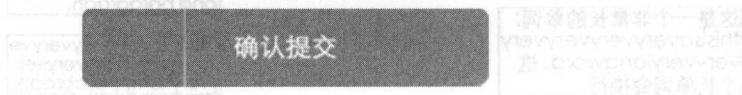


图 4.47 圆角按钮效果

3. border-image

border-image 用于设置或检索对象使用图像来填充。之前边框只允许用颜色来填充，CSS 3 开始引入边框图片填充，这是一个重大的突破。**border-image** 属性基本语法如下：

```
border-image: <'border-image-source'> || <'border-image-slice'> [ / <'border-image-width'>
| / <'border-image-width'>? / <'border-image-outset'> ]? || <'border-image-repeat'>;
```

border-image 属性是一个简写属性，可以被拆解为以下 5 个属性。

- **border-image-source**: 用在边框的图片的路径。
- **border-image-slice**: 图片边框向内偏移。
- **border-image-width**: 图片边框的宽度。
- **border-image-outset**: 边框图像区域超出边框的量。
- **border-image-repeat**: 图像边框是否应平铺 (repeated)、铺满 (rounded) 或拉伸 (stretched)。

使用 Chrome 浏览器打开 **border-image.html** 实例文件，效果如图 4.48 所示。

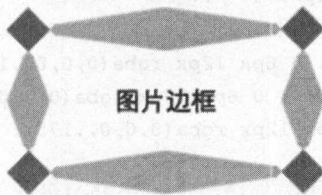


图 4.48 图片边框效果

4. box-shadow

box-shadow 属性用于向边框添加一个或者多个阴影，语法如下：

```
box-shadow: h-shadow v-shadow blur spread color inset;
```

取值说明如下。

- **h-shadow**: 必需，水平阴影的位置，允许负值。
- **v-shadow**: 必需，垂直阴影的位置，允许负值。
- **blur**: 可选，模糊距离。
- **spread**: 可选，阴影的尺寸。
- **color**: 可选，阴影的颜色。请参阅 CSS 颜色值。
- **inset**: 可选，将外部阴影 (outset) 改为内部阴影。

提示: box-shadow 向框添加一个或多个阴影。该属性是由逗号分隔的阴影列表, 每个阴影由 2 至 4 个长度值、可选的颜色值以及可选的 inset 关键词来规定。省略长度的值为 0。

使用 box-shadow 制作下拉框和菜单的阴影, 代码如下:

```

01 <html>
02 <head>
03     <meta charset="UTF-8">                                /* 页面字符编码 */
04     <title>box-shadow</title>                             /* 页面标题 */
05     <style type="text/css">
06         .dropdown-menu {
07             display: block;                                /* 块级元素 */
08             width: 200px;                                  /* 元素宽度 */
09             list-style: none;                              /* 无序列表样式 */
10             border: 1px solid #ccc;                       /* 边框 */
11             border-radius: 4px;                            /* 边框圆角 */
12             padding: 0;                                    /* 元素内边距 */
13             -moz-box-shadow: 0 6px 12px rgba(0,0,0,.175); /* Firefox */
14             -webkit-box-shadow: 0 6px 12px rgba(0,0,0,.175); /* Chrome */
15             box-shadow: 0 6px 12px rgba(0,0,0,.175);      /* 阴影标准写法 */
16         }
17         .dropdown-menu li {
18             font-size: 14px;                                /* 字体大小 */
19             width: 200px;                                  /* li 元素宽度 */
20             height: 40px;                                  /* li 元素高度 */
21             line-height: 40px;                             /* 行高 */
22             text-align: center;                             /* 文字居中 */
23             border-bottom: 1px solid #ccc;                 /* 下边框 */
24         }
25         .dropdown-menu li:last-child {                    /* 匹配最后一个 li */
26             border-bottom: none;                            /* 无下边框 */
27         }
28         .dropdown-menu li a {
29             color: #333;                                    /* 字体颜色 */
30             text-decoration: none;                          /* 去除下划线 */
31         }
32     </style>
33 </head>
34 <body>
35     <ul class="dropdown-menu">

```

```

36         <li><a href="#">菜单一</a></li>
37         <li><a href="#">菜单二</a></li>
38         <li><a href="#">菜单三</a></li>
39         <li><a href="#">菜单四</a></li>
40     </ul>
41 </body>
42 </html>

```

使用 Chrome 浏览器打开 box-shadow.html 实例文件，查看阴影菜单效果，如图 4.49 所示。

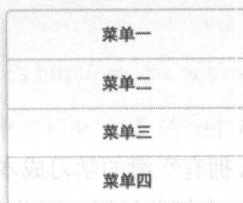


图 4.49 阴影菜单效果

4.6 预编译

CSS 作为一门样式设计语言，其语法相对简单，对使用者要求较低，因此很容易编写出大量看似没有逻辑、难以扩展且不利于复用的代码，这是因为 CSS 并非一门程序式语言。所以在开发者日常使用中，如果没有完善的编码规范，编写的 CSS 代码会十分冗余且较难维护。

CSS 语言主要缺乏的语言特性有：变量、运算、循环、函数、作用域。开发者为了克服这些问题，制造了多种预编译语言。CSS 预编译语言是一种基于 CSS 语言的语法扩展，除了能解决以上缺乏语言特性带来的问题以外，还支持嵌套书写，减少重复输入父级选择器，便于编写和阅读代码。常见的预编译语言主要有 Less、Sass、Stylus 等。本章主要介绍 Less 和 Sass 这两门语言来展示 CSS 预编译语言的环境配置和使用。

4.6.1 Less 介绍和安装

Less 是由 Alexis Sellier 在 2009 年开发的一款 CSS 预编译语言，以“.less”作为文件的后缀名，官网地址为 <http://lesscss.org/>。Less 的成功之处在于其语法特性，同时期的 Sass 的一个关键特征是缩进式语法（具体会在 Sass 的介绍中说明），而 Less 风格则继承 CSS 的花括号，所以每一个语法正确的 CSS 代码也是合法的 Less 代码。

Sass 缩进式语法, 代码如下:

```
h1
  margin: 10px 0
  font-size: 24px
  color: #333
```

CSS 和 Less 花括号语法, 代码如下:

```
h1 {
  margin: 10px;
  font-size: 24px;
  color: #333;
}
```

相对于别的 CSS 预编译语言, Less 拥有平滑的学习成本, 语法更接近于 CSS 语法, 对于开发者来说, 完全可以把 Less 看作 CSS 的一个简单扩展, 而并非一门全新的语言。

Less 的最早版本采用 Ruby 语言实现, 当前版本已经换为 JavaScript, 对于前端开发者来说, 安装和使用更为熟悉, 使用方式主要有以下几种。

1. 命令行使用

命令行用法也是服务器端用法, 需要开发者安装 Node.js, 并通过 NPM 来安装 Less 编译器, 命令如下:

```
$ npm install less -g
```

现在可以通过 lessc 命令进行编译, 命令如下:

```
/* option 为可选参数, source 为源文件(less 文件), destination 为编译后文件(css 文件) */
$ lessc [option option=parameter ...] <source> [destination]
```

Less 文件可以直接通过命令行进行编译, 命令如下:

```
$ lessc style.less style.css // 将 style.less 文件编译成 style.css
```

添加“-x”参数, 对编译后的文件进行压缩, 命令如下:

```
$ lessc -x style.less style.css // 将 style.less 文件编译成压缩过的 style.css
```

2. 浏览器使用

Less 为开发者提供了一个非常便捷的功能, 支持在浏览器端进行编译执行。可以前往官网下载客户端脚本, 或者直接使用官方 CDN 数据源, 代码如下:

```
<link rel="stylesheet/less" type="text/css" href="/path/to/style.less">
```

```
<script src="//cdnjs.cloudflare.com/ajax/libs/less.js/2.7.2/less.min.js"></script>
```

在引入“less.js”文件前，通过定义全局的 less 对象设置参数，代码如下：

```
<script>
  less = {
    env: "development",           // 相关信息会被输出到控制台
    logLevel: 1                   // 只有错误信息才会被输出
    globalVars: { myvar: "#ddffee" }; // 全局变量
  }
</script>
```

更多的参数设置，请参考官方文档 <http://lesscss.org/usage/#using-less-in-the-browser-options>。

注意：在引入的“.less”文件中，link 元素的 rel 属性要设置为“stylesheet/less”，并且“.less”文件一定要在“less.js”引入之前引入，这样才能保证 Less 文件被正确编译解析。不过，在浏览器端编译 Less 的做法并不是被推荐的做法，因为 Less 最终需要被编译成 CSS 才能被浏览器识别。当 Less 在浏览器端编译成 CSS 时，用户会看到延迟，并且“less.js”需要浏览器支持 ES5 语法，在某些浏览器下有可能引发 JavaScript 报错。

3. 在线 Less 编译器

开发者在学习入门阶段，可以通过在线 Less 编译器快速查看 Less 代码所对应的 CSS，可以使用的网站有：

- <http://less2css.org>
- <http://winless.org/online-less-compiler>

4. 图形界面的 Less 编译器

本地图形界面 Less 编译器也是一种很适合入门使用的方式，处理文件更加方便直观，例如软件 SimpLESS，开发者只需要拖曳文件到界面中，选择编译即可，如图 4.50 所示。

提示：SimpLESS 官方下载地址为
<http://wearekiss.com/simpless>。

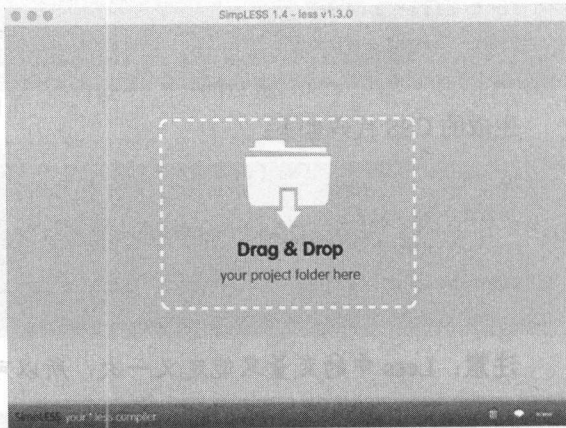


图 4.50 SimpLESS 界面

4.6.2 Less 使用

本节将从变量、嵌套、作用域、混合、约束与循环、导入等几个方面,介绍 Less 在样式开发中提供的强大功能。

1. 变量

Less 提供了变量功能,这意味着开发者不需要重复编写相同的 CSS 属性,变量的使用十分简单便捷,Less 的变量通过“@”符号作为标记,代码如下:

```
// 为多个选择器设置同一个颜色
@primaryColor: #44B336;
h1 { color: @primaryColor; }
.title { color: @primaryColor; }
```

生成的 CSS 代码如下:

```
h1 { color: #44B336; }
.title { color: #44B336; }
```

Less 中除了普通变量外,还有允许插值变量。插值变量可以让变量用在 CSS 规则以外的地方,包括选择器名、属性名等,代码如下:

```
// 用变量生成选择器名
@mySelector: title
. @{mySelector} {
  color: #44B336;
  font-size: bold;
  line-height: 40px;
}
```

生成的 CSS 代码如下:

```
.title {
  color: #44B336;
  font-size: bold;
  line-height: 40px;
}
```

注意: Less 中的变量只能定义一次,所以可以被认为是编程语言中的常量。

2. 嵌套

过往的开发中,开发者常常不可避免写出这样的 CSS,代码如下:

```
#header { display: inline; float: left; }
```

```
#header h1 { font-size: 24px; font-weight: bold; }
#header h1 a { color: #44B336; }
```

这样的写法意味着开发者需要重复在代码中编写冗余的选择器，而 Less 提供了嵌套来解决这样的问题，同时让选择器之间的逻辑看起来更加清晰，代码如下：

// 用嵌套展示清晰的逻辑关系

```
#header {
  display: inline;
  float: left;
  h1 {
    font-size: 24px;
    font-weight: bold;
    a { color: #44B336; }
  }
}
```

嵌套一个更强大的功能是提供了“&”运算符表示父选择器，这个运算符提供了两个很实用的功能，一是可以用来应用伪类，Less 代码如下：

```
a {
  color: #333;
  &:hover { color: #44B336 }
}
```

生成的 CSS 代码如下：

```
a { color: #333; }
a: hover { color: #44B336; }
```

二是可以用来生成重复的类名，Less 代码如下：

```
// 多个选择器共用同一个选择器作为选择器名的一部分
.button {
  &-ok { background-color: green; }
  &-warn { background-color: yellow; }
  &-danger { background-color: red; }
}
```

生成的 CSS 代码如下：

```
.button-ok { background-color: green; }
.button-warn { background-color: yellow; }
.button-danger { background-color: red; }
```

3. 作用域

Less 的作用域理解起来很简单, 因为变量可以被视为一种“常量”, 所以 Less 的作用域主要体现在局部变量和全局变量之上, 即查找变量的顺序是先在局部定义中, 如果找不到, 则会向上一级, 直到全局。这里有一个简单的例子来体现作用域, 代码如下:

```
@primaryColor: #44B336;
#header {
  @primaryColor: #333;
  h1 { color: @primaryColor }           // color 值为#333
}
#footer { color: @primaryColor; }       // color 值为#44B336
```

4. 混合

Less 中可以将任意的“类”选择器和“id”选择器混合到另外一个选择器之中, 而被嵌入的这个选择器也可以被视为一个变量。被混入的部分在 Less 中称为混合集, 这种做法可以减少重复的属性书写, 也可以视为 Less 提供的一种抽象方法。

选择器作为混合集是 Less 灵活且接近 CSS 语法的体现, 即混合集并非特殊的存在。但有些情况下开发者并不想将混合集的内容输出到最终的 CSS 的样式中。Less 提供了一种解决方案, 在混合集的名字后面加上一对括号, 这种混合集可以被视为一个函数声明, 同时这种混合集可以接受参数, 提供更好的复用功能。

用于兼容不同浏览器 border 样式的前缀, Less 代码如下:

```
/* 定义一个规则, 并且不设置默认参数值 */
.borderRadius(@radius) {
  -moz-border-radius: @radius           // 兼容 firefox
  -webkit-broder-radius: @radius        // 兼容 chrome
  border-radius: @radius
}
/* 应用到元素中 */
#header { .borderRadius(10px); }        // 将 10px 传给变量@radius
.btn { .borderRadius(3px) }
```

生成的 CSS 代码如下:

```
#header {
  -moz-border-radius: 10px;
  -webkit-border-radius: 10px;
  border-radius: 10px;
}
```

```
.btn {
  -moz-border-radius: 3px;
  -webkit-border-radius: 3px;
  border-radius: 3px;
}
```

5. 约束与循环

对于 Less 来说为了保持与 CSS 接近的语法, 并没有引入编程语言中常见的“if/else”语句, 而是选择了约束来匹配表达式。Less 通过 `when` 关键字实现约束功能, 约束在 Less 的早期版本只能在混合 (Mixin) 上, 可以视为对 Mixin 参数的一种扩展, 现在也可以直接被用在 CSS 选择器上。

同样, Less 并没有引入类似 `for` 的关键词来实现循环。Less 通过混合功能, 可以调用自身的特性再结合约束, 让开发者可以实现出循环的结构。一种常见的应用就是生成栅格系统, 代码如下:

```
.generate-column(4);
.generate-column(@n, @i:1) when(@i <=@n) {           // n, i 是 mixin 的两个参数, i<=n 是条件
  .column-@{i} {                                         // 用 i 作为插值变量
    width: (@i * 100% / @n);                             // 运算生成 width 的宽度
  }
  .generate-column(@n, (@i + 1));                       // 将 i+1 递归调用自身 mixin
}
```

生成的 CSS 代码如下:

```
.column-1 { width: 25%; }
.column-2 { width: 50%; }
.column-3 { width: 75%; }
.column-4 { width: 100%; }
```

6. 导入

导入功能在原生 CSS 中就有相应的实现, 但在实际开发过程中并不推荐开发者使用。原生 CSS 通过“`@import`”关键字实现导入功能存在两大弊端, 首先, “`@import`”导入规则必须先于除了“`@charset`”外的其他 CSS 规则执行。其次, 导入的文件需要等待引用其父文件下载解析才被执行, 如此, 丢失了 CSS 并行下载的能力, 加大了资源加载的开销。

Less 提供的导入方法是在编码阶段被引入的, 在最终代码中合并成单个 CSS 文件。除此之外, 还提供了导入选项来更灵活地引入第三方文件。

4.6.3 Sass 介绍和安装

Sass 是由 Hampton Catlin 在 2007 年由 Hampton 设计并由 Natalie Weizenbaum 开发的层叠样式表语言。不同于 Less 的是, Sass 具有两种不同的后缀名分别对应两套语法, 最早 Sass 使用的是缩进式语法, 使用缩进来区分代码块, 并使用分号将具体的样式分开, 这种语法使用“.sass”作为后缀, 另一种使用了和 CSS 一样的块语法, 这种语法使用“.scss”作为后缀。

本书主要介绍 SCSS 语法, 虽然 Sass 语法更加简练, 但是并不兼容原生 CSS 语法, 对于初学者来说, SCSS 语法接近原生 CSS 的面貌使之更易上手, 更像是一种 CSS 的语法扩展, 也更受开发者的欢迎。值得注意的是, Sass 和 SCSS 之间可以互相导入, 并且可以通过“sass-convert”命令行工具进行切换。

Sass 的官方解释器使用 Ruby 语言编写, 同时也有其他语言实现的版本, 比较常用的是 C 语言实现的 libSass, 为了方便使用各种 Sass 扩展, 推荐首先安装 Ruby 环境。

1. Ruby 的安装

• Windows 环境安装

在 Windows 环境下推荐使用 RubyInstaller。安装完成后, 在命令行中执行如下命令:

```
ruby -v
```

如果正常显示版本号, 表示安装成功, 如果没有正常显示, 则可能缺少配置系统变量。

• Mac 环境安装

Mac 环境推荐使用 Homebrew, 首先执行以下命令完成 Homebrew 的安装, 如果已安装可以跳过。安装 Homebrew, 命令如下:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

然后通过 brew 命令安装 Ruby, 命令如下:

```
brew update
brew install ruby
```

安装成功后, 执行“ruby -v”命令确定已成功安装最新版本的 Ruby。

• Linux 环境安装 (Debian 或 Ubuntu)

Linux 环境下可以选择使用发行版系统的包管理器进行安装, 以 Debian 或 Ubuntu 的 apt 为例, 命令如下:

```
apt-get install ruby
```

安装成功后，执行“`ruby -v`”确定已成功安装最新版本的 Ruby。

2. Sass 的安装

对于一般的前端开发者，Sass 有两种主流的开发方式，一种可以通过 Ruby 的包管理工具 Gem 安装 Ruby 编译版本的 Sass。命令如下：

```
gem install sass
```

另一种是可以通过 NPM 安装 Node.js 包装的 C 语言版本 libSass（需要配置 C 编译环境）。

```
npm install node-sass // 安装 node-sass
```

注意：在 Windows 环境下需要安装 Visual Studio 或者 GCC 提供 C 语言编译环境。

3. 命令行使用

如果是通过 Gem 安装的 Sass，可以使用命令行进行 Sass 文件的编译，命令如下：

```
sass style.scss style.css
```

Sass 命令还提供了监视文件甚至整个目录改动的功能。开发者对 Sass 文件的修改会即时更新到对应的 CSS 文件中，命令如下：

```
sass -watch style.scss:style.css // 监视单个文件
sass -watch src/styles:public/styles // 监视整个文件夹
```

4. 通过 JavaScript 使用

如果安装的 node-sass，可以选择通过构建工具进行自动化构建或者直接通过 JavaScript 编译 Sass 文件，代码如下：

```
var sass = require('node-sass'); // 导入 node-sass 模块
sass.render({
  file: sass_filename,
  [, option..] // 一些 node-sass 的配置
}, function(err, result) { /*...*/ }); // 可以对获取的结果再进行处理
```

5. 图形界面的 Sass 编译器

为了方便开发者能够快速上手 Sass，图形界面的 Sass 编译器推荐使用 Koala。Koala 是一款跨平台的免费 Sass 编译工具。下载安装完后可以快速拖动项目到 Koala 面板中，如图 4.51 所示。

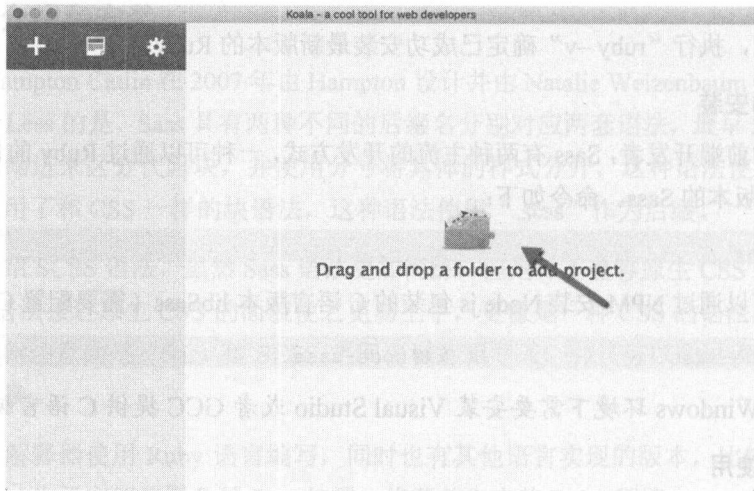


图 4.51 Koala 界面

之后当开发者创建或保存 Sass 文件时，Koala 都会自动地完成编译过程。

4.6.4 Sass 使用

Sass 提供了变量、嵌套、混合、导入、循环等功能，在调用方法和具体功能上和 Less 等预编译语言有一些区别，本节会简单介绍使用上的区别，着重突出 Sass 的特色功能。Sass 和 Less 等其他预编译语言并不存在优劣之分，一般来说 Sass 的功能更加强大或者说 Sass 的语言层面更接近于一门完整的语言，而 Less 则更亲近于 CSS 语法。

1. 变量

Sass 的变量通过 “\$” 作为标识符，除了简单的单值变量，Sass 还提供了列表和 Map，列表是一个一维的数组，代码如下：

```
$font-stack: ('Helvetica', 'Arial', sans-serif);
```

列表中的值可以通过 nth 方法传入变量名和要取值的索引来获得，注意在 Sass 中索引的起始值是 1 而不是 0。

```
nth($font-stack, 0)    // 报错  
nth($font-stack, 1)    // Helvetica
```

Map 是一种更复杂的数据结构，可以映射任何类型的键值对，甚至可以 Map 间相互嵌套。一个简单 Map 的变量代码如下：

```
// 媒体查询需要的断点值
$breakpoints: (
  small: 767px,
  medium: 992px,
  large: 1200px
);
```

Sass 提供了一个 `map-get` 方法来获取 Map 中的值，代码如下：

```
// 生成媒体查询，可以多次复用$breakpoints
@media (min-width: #{map-get($breakpoints, small)}) {...}
```

上面的实例还展示了 Sass 变量的另一个特点，当变量作为插值变量（即变量作为选择器和属性名）的时候需要用“#{ }”作为标示，代码如下：

```
$name: foo;
$attr: border;
p.#{ $name } { #{ $attr }-color: #44b336; }
```

2. 混合

Sass 的混合（Mixin）并不像 Less 一样完全贴近 CSS 语法，开发者需要用“@mixin”标记来定义 Mixin，也就是说不能将任意一个选择器视为 Mixin，需要为 Mixin 单独命名。Mixin 也拥有完整的功能，Mixin 可以包含多个选择器，甚至可以包含尚未设定父元素的“&”选择符来指代将被混入的父元素。

注意：在介绍 Sass 的混合前需要注意一点，Sass 支持所有的 CSS 3 的“@”规则（比如媒体查询），同样 Sass 的很多特殊指令也会用“@”作为标记用来与一般的 CSS 代码进行区别，对于本节后面出现的“@”开头的指令，不对“@”标识符再做解释。

Sass 的 Mixin 另外一个强制规范是需要通过“@include”来混入一个 Mixin，代码如下：

```
// 清除浮动元素
@mixin clearfix {
  zoom: 1;
  &: before,
  &: after { content: ""; display: table; }
  &: after {
    clear: both;
    visibility: hidden;
    font-size: 0;
    height: 0;
  }
}
```



```

    }
  }
  div { @include clearfix; } // 使用 mixin

```

3. 函数

在 Sass 中, Mixin 更注重的是对重复 CSS 代码的抽象, 而当需要对变量进行复杂处理的地方, 可以使用 “@function” 来定义一个方法进行处理。Sass 文件代码如下:

```

// 栅格布局
$grid-width: 40px; // 默认每格宽度
$gutter-width: 10px; // 默认每格间距
@function grid-width($n) {
  // 可以为不同元素设置不同的格数, 间距保持统一
  @return $n * $grid-width + ($n - 1) * $gutter-width;
}
// 生成一个占据五格的 sidebar
#sidebar { width: grid-width(5); }

```

生成的 CSS 文件, 代码如下:

```

$sidebar {
  width: 240px;
}

```

函数在 Sass 中可以用来处理重复的计算, 或者将一些元变量直接拼接成一个新的变量。不同于 Mixin 的是, 函数的使用在 Sass 中不需要特殊的指令来表示。

4. 循环与条件语句

从某种层面来看 Sass 可以被认为是一门完整的脚本语言。在循环与条件上, Sass 与一般的编程语言使用方法十分类似, 提供了 “@if”、“@for”、“@each”、“@while” 等指令用于这个功能。

Sass 在 “@if” 上同样提供了 “@else if” 和 “@else”, 开发者可以在任何选择器中进行判断, 代码如下:

```

$type:huijiang;
// 根据 type 变量生成 p 选择器的颜色
p {
  @if $type == baidu {
    color: blue;
  } @else if $type == alibaba {
    color: orange;
  }
}

```

```

} @else if $type == hujiang {
    color: green; // 因为 type 为 hujiang, color 为 green
} @else {
    color: black;
}
}

```

“@for”指令可以用来生成一系列的样式，在每一次循环中，一个计算值作为变量来控制输出。Sass 文件代码如下：

```

@for $i from 1 through 3 {
    .item-#{ $i } { width: 2em * $i; } // 从 item-1 到 item-3 每个增加 2em 宽度
}

```

生成的 CSS 文件，代码如下：

```

.item-1 { width: 2em; }
.item-2 { width: 4em; }
.item-3 { width: 6em; }

```

除了“from...through”，“@for”指令也可以使用“from...to”。两者的区别在于 through 的循环中包含 end 值，而 to 的循环中不包含。另一点需要注意的是 start 和 end 需要是整型值，但是 end 可以大于 start，这种情况下会采用降序计算而非升序计算。

当需要遍历列表或者 Map 的时候，可以选用“@each”直接获取数据结构中的值。Sass 代码如下：

```

@each $header, $size in (h1: 2em, h2: 1.5em, h3: 1.2em) {
    #{ $header }
    font-size: $size // 快速生成不同字号的选择器
}

```

生成的 CSS 代码如下：

```

h1 { font-size: 2em; }
h2 { font-size: 1.5em; }
h3 { font-size: 1em; }

```

本节主要介绍了 Sass 的使用方法，因为已经对 Less 的语法进行了介绍，所以在本节就不再重复部分的语法基础。Sass 很受欢迎的一个特点在于有丰富的混合集可以被引入使用，在下一节中会介绍一个 Sass 的工具 Compass。

4.6.5 Compass 的安装和使用

Sass 作为一门 CSS 预编译语言可以使 CSS 的开发变得简单和可维护,但如果要体现出 Sass 强大的功能,需要配合 Compass 使用。

Compass 是 Sass 的工具库,在 Sass 的基础上,封装了大量实用的模块,使开发者无须实现一些函数和混合集,而是通过 Compass 直接调用即可。对于前端开发者来看,Sass 和 Compass 的关系类似于 JavaScript 和 jQuery 的关系。

1. 安装 Compass

安装好 Ruby 之后,在命令行中安装 Compass,命令如下:

```
gem install compass
```

2. 项目中引入 Compass

如果项目尚未建立,可以通过 Compass 创建一个项目,命令如下:

```
compass create <myproject>
```

如果项目已经建立,可以切换到项目目录初始化 Compass,代码如下:

```
cd <myproject>
compass init
```

Compass 初始化完成后,在项目目录下会有一个 config.rb 文件,这个文件是项目的配置文件。还有两个子目录 sass 和 stylesheets。sass 目录用来存放源文件,stylesheets 目录存放编译后的 CSS 文件。

3. Compass 编译

Compass 的编译类似于 Sass 的编译命令,命令如下:

```
compass compile
```

这个命令会将 sass 目录下的文件编译为 CSS 文件,并保存至 stylesheets 目录下。

Compass 编译提供了一些可选参数,命令如下:

```
compass compile --output-style compressed // 编译成压缩文件
compass compile -force // 编程全部文件,默认只编译改动过的文件
```

跟 Sass 编译相同,Compass 也有自动编译功能,命令如下:

```
compass watch
```

4. Compass 模块使用

Compass 包含五个内置模块，下面将分别介绍这五个内置模块。

- Reset

开发者可以通过“@import”指令导入模块。通常，开发者需要重置浏览器的默认样式，此时可以通过引入 Reset 模块来解决，代码如下：

```
@import "compass/reset"
```

- CSS 3

CSS 3 模块，可以让开发者不必再关心浏览器对 CSS 3 的兼容前缀问题，比如需要设置一个圆角，在引入 CSS 3 模块之后，开发者只需要这样编写代码：

```
@import "compass/CSS 3"
.rounded { @include border-radius(5px); }
```

生成的 CSS 代码如下：

```
.rounded {
  -moz-border-radius: 5px;           //兼容 firefox
  -webkit-border-radius: 5px;       //兼容 chrome
  border-radius: 5px;
}
```

Compass 还提供了 border-conner-radius、border-top-left-radius、border-top-right-radius 等方便开发者设置圆角细节，同时还支持包括透明度、过滤、Font Face、行内区块等。

- Layout

Layout 模块主要提供了布局上的功能，这个模块使用率较低，提供了三个子模块：stretch-full、sticky-footer 和 grid-background。

stretch-full 模块可以生成相对父元素的绝对定位，Sass 代码如下：

```
.stretch-full { @include stretch(); }
```

生成的 CSS 代码如下：

```
.stretch-full {
  position: absolute;
  top: 0;
  bottom: 0;
  left: 0;
  right: 0
}
```


sticky-footer 模块可以快速生成一个固定在浏览器底部的 Footer 区域, Sass 代码如下:

```
#footer {  
  @include sticky-footer(40px);           // 参数为 footer 的高度  
}
```

grid-background 模块可以为元素添加栅格背景。

- **Typography**

Typography 主要提供版式功能, 比如, 开发者要制定链接在不同状态下的链接颜色, Sass 代码如下:

```
//不同状态下的超链接颜色  
@import "compass/typography/links"  
a { @include link-colors(#333, #999, #44b336, #44b336 - #111, #44b336 + #111 )
```

生成的 CSS 代码如下:

```
// 不同状态下的超链接颜色  
@import "compass/typography/links"  
a { color: #333;}  
a: visited { color: #999;}  
a: focus { color: #44b336;}  
a: hover { color: #43b235;}  
a: active { color: #45b437;}
```

- **Utilities**

Utilities 提供了剩余的模块功能, 最常见的需求是清除浮动, 开发者只需要一行代码就可以解决这个问题, 代码如下:

```
.clearfix { @include clearfix }
```

Utilities 提供的另外一个强大功能是精灵图, 在网站开发的时候, 开发者会合并部分小图片到一张大图上, 再利用 CSS 的 **background-position** 属性定位每个小图片在大图上的相对位置。这样做可以减少大量的请求, 而 **Utilities** 模块提供了自动合成精灵图的功能, 开发者不需要再手工计算每张图的相对位置, 调用十分简单, 代码如下:

```
@import "compass/utilities";  
@import "sp/*.png";           // 导入 sp 目录下所有 png 图片  
@include all-share-sprites    // 输出所有的雪碧图 CSS
```

进行编译后, 在 **images** 目录下会出现一张合成的雪碧图并且生成对应的 CSS 文件, 生成的 CSS 代码如下:

```

.sp-img1, .sp-img2, sp-img3 {
    background-image: url('../images/sp.png');
    background-repeat: no-repeat;
}
.sp-img1 { background-position: 0 0; }
.sp-img2 { background-position: 0 -20px; }
.sp-img3 { background-position: 0 -40px; }

```

4.7 本章小结

CSS 3 发展到现在, 主流浏览器对其中很多重要特性的支持度已经越来越高。尤其是在不需要考虑低端 Internet Explorer 浏览器的移动 Web 前端, CSS 3 的很多重要特性已经被大量应用到移动站点中。可以说学习和使用 CSS 3 已经成为进行移动 Web 前端开发不可或缺的一部分。本章在介绍了 CSS 3 相关的常用特性之外, 着重讲述了 CSS 3 在布局、动效方面的使用。另外, 本章还详细介绍了 CSS 3 在浏览器兼容性方面的相关知识点。读者结合这些内容, 就可以在移动 Web 前端站点中放心大胆地使用 CSS 3 来提高开发效率和站点性能。

5

第 5 章

JavaScript 关键语法及使用技巧

JavaScript 从诞生之初一直是门颇具争议的语言，伴随着浏览器端的崛起，JavaScript 成为了现今最红的语言之一。直到今天，谈起 JavaScript 已不局限于浏览器内部，在后端、手机原生端、硬件层面都可以看到 JavaScript 的身影。如果你是一名前端工程师，就更应该充分掌握这门语言。

本章将从 JavaScript 的早期版本开始，介绍该语言关键的语法点和使用技巧，帮助读者更快速地掌握。

5.1 理解 JavaScript

JavaScript 是浏览器所支持的一种脚本语言，是 ECMAScript 语言的一种实现，基于原型、多范式的动态脚本语言，并支持面向对象、命令式和声明式（如函数式编程）编程风格。JavaScript 包括 DOM（文档对象模型）和 BOM（浏览器对象模型）。本节介绍内容包含语言基础、函数和参数这三个基本部分。

5.1.1 语言基础

本节介绍 JavaScript 语言的基础，为学习后面的知识做好铺垫。

1. 变量

JavaScript 中使用 var 关键字定义变量，例如：

```
var m = 1;
```

上例中的 m 是变量名，变量名的命名规则是由字母、数字、\$、下划线组成，但不能以数字开头，并且区分大小写。变量是动态类型的，可再次赋值其他数据类型。

2. 数据类型

JavaScript 中的数据类型包括数字、字符串、布尔、数组、对象、Null、Undefined。

- 数字：包括整数和小数，例如 5、0.2、-3 等。
- 字符串：由单引号或双引号包围起来的一段字符，代码如下：

```
var title = 'Hello World';  
var text = "JavaScript 是运行在浏览器中的脚本语言";
```

- 布尔：true 和 false，表示逻辑真和假。
- 数组：表示一组数据，用中括号包围，逗号进行分隔，代码如下：

```
var fruitList = ['apple', 'orange', 'banana'];
```

提示：JavaScript 数组中的每个元素数据类型允许不相同，同时数组的长度可以动态变化。

- 对象：由键值对（name:value）组成的数据类型，用花括号包围，逗号分隔，代码如下：

```
var user = { name: 'tom', age: 12 };
```

有两种方式可以访问对象的属性，代码如下：

```
console.log(user.name);  
console.log(user['name']);
```

- Null 和 Undefined：Null 表示空值，Undefined 表示未定义。

3. 运算符

JavaScript 运算符用于赋值、比较、执行算术运算等。常用运算符见表 5.1。

表 5.1 JavaScript运算符

类 型	运 算 符	描 述	例
算术运算符	+	加	x=y+1
算术运算符	-	减	y=x-1
算术运算符	*	乘	i=j*3
算术运算符	/	除	j=i/3
算术运算符	%	取模	k=i%2
算术运算符	++	累加	x++
算术运算符	--	递减	y++
赋值运算符	=	赋值	x=y+1
赋值运算符	+=	加	x+=1, 即x=x+1
赋值运算符	-=	减	y-=1, 即y=y-1
赋值运算符	*=	乘	i*=3, 即i=i×3
赋值运算符	/=	除	j/=3, 即j=j/3
赋值运算符	%=	取模	k%=2, 即k=k%2
比较运算符	==	等于	x==y
比较运算符	===	全等于	x===y
比较运算符	!=	不等于	x!=y
比较运算符	>	大于	x>y
比较运算符	<	小于	x<y
比较运算符	>=	大于等于	x>=y
比较运算符	<=	小于等于	x<=y
逻辑运算符	&&	与	a&& b
逻辑运算符		或	a b
逻辑运算符	!	非	!a
连接运算符	+	连接字符串时	'hello'+'world'

4. 条件

JavaScript 中的条件语句有如下几种形式。

- “if” 语句：条件为 true 时执行代码，代码如下：

```
if (a > b) { return a - b; }
```

- “if...else” 语句：条件为 true 时执行代码，否则执行 else 代码块，代码如下。

```
if (a > b) { return a - b; } else { return b - a; }
```

- “if...else if...else” 语句：多个条件时，选择条件为 true 的代码块执行，如果没有合适匹配直至执行 else 代码块，代码如下：

```
if (score >= 60 && score < 80) {  
    return '及格';  
}
```

```

} else if (score >= 80 && score < 90) {
    return '良';
} else if (score >= 90) {
    return '优秀';
} else {
    return '不及格';
}

```

- switch 语句：多个条件时，选择执行对应动作，代码如下：

```

switch(fruit) {
    case 'banana': return 'yellow';
    case 'apple': return 'red';
    default: return 'green';
}

```

5. 循环

使用循环语句可以很方便地重复执行某段代码，JavaScript 中的循环语句包括以下几种。

- “for” 语句：适合执行某段代码按指定次数完成，代码如下：

```
for (var i = 0; i < 10; i++) { console.log(i); } // 在控制台循环输出数字 0 到 9
```

- “for...in” 语句：遍历数组的元素或者对象的属性，代码如下：

```

for(var n in [1, 2, 3]) { console.log(n); } // 循环输出数组元素 1, 2, 3
for(var p in { a: 1, b: 2 }) { console.log(p); } // 循环输出对象属性 a, b

```

- “while” 语句：根据循环条件判断是否继续执行循环体内的代码，代码如下：

```

var a = 0, b = 50;
while( a < b ) { // 当 a<b 的时候执行循环体中的代码，直到 a 大于等于 b 的时候循环结束
    a += 10; // a 累加 10
    b += 5; // b 累加 5
}

```

- “do...while” 语句：和 while 语句类似，但会先执行一次循环体再根据循环条件判断是否继续执行，代码如下：

```

// 先执行一次循环体中的代码，再判断 a 是否小于 b，满足条件继续执行
do {
    a += 10;
    b += 5;
} while ( a < b )

```

6. 函数

函数是一段指定名称的代码块, 可接收参数, 返回运行结果, 语法如下:

```
function 函数名(参数 1, 参数 2, ...) { 代码块 }
```

实现将两个数字相加功能, 代码如下:

```
function add(m, n) { return m + n; }
var sum = add(3, 2);
```

上面的例子中, 定义了一个名为 `add` 的函数, 函数接收两个参数 `m` 和 `n`, 计算 `m` 与 `n` 的和, 并通过关键字 `return` 返回结果。调用方法的时候按照函数定义时的参数顺序传递参数。

7. 异常

异常捕获处理语法如下:

```
try { 代码块 } catch (error) { 异常处理代码块 }
```

`try` 后面的代码块是正常的业务代码, 执行过程中可能会抛出某个异常, `catch` 后面的 `error` 是捕获到的异常信息。异常处理代码块中可对捕获到的异常信息进行处理, 代码如下:

```
function getLength(str) {
  try {
    return str.length;
  } catch (error) {
    console.log('参数 str 不是有效的字符串');
    return 0;
  }
}
```

上面的例子中, `getLength` 函数用于获取传入字符串的长度, 但当调用函数时传递的参数为 `null` 或者是其他非字符串类型时, `str.length` 会抛出异常。此处, 通过 `catch` 捕捉异常, 并返回一个默认值 `0`。

JavaScript 的异常机制, 可以捕获代码执行过程中出现的错误, 并对异常进行处理, 避免程序中断执行而影响用户操作, 同时可对用户进行合理有效的提示。

5.1.2 函数和参数

函数是任何语言中都非常重要的一部分。官方的解释是, 函数是由事件驱动的或者当它被调用时执行的可重复使用的代码块。本节会详细的介绍函数及参数的使用。

1. 函数声明

• 函数式

前面曾介绍使用 `function` 关键字声明一个函数，代码如下：

```
function add(m, n) { return m + n; }
```

同样，也可以通过一个表达式来声明函数，把函数指定给一个变量，通过变量名调用该函数，代码如下：

```
var sum = function add(m, n) { return m + n; }
var result = sum(1, 2);
```

• 变量式

函数也是一个对象，可以通过创建一个 `Function` 类的实例来声明一个函数，语法如下：

```
var 方法名 = new Function(参数1, 参数2, ..., 参数n, 函数体)
```

使用变量式创建一个求和的函数，代码如下：

```
var add = new Function('m', 'n', 'return m + n;')
```

2. 匿名函数

声明函数时，省略函数的名字即为匿名函数。因为省略了函数名，无法直接按名字调用执行，所以通常匿名函数的使用可以指定给一个变量或者直接通过传参执行。

把匿名函数指定给一个变量，代码如下：

```
var add = function (m, n) { return m + n; };
```

匿名函数立即执行，代码如下：

```
var result = (function(m, n) { return m + n; })(1, 2);
```

匿名函数作为参数传递，代码如下：

```
document.addEventListener('click', function(e){ console.log(e); });
```

3. 函数的参数

函数可以通过参数来接收外部数据，声明函数时可以指定任意个参数，在函数体内可以通过参数名进行调用。

• 参数可选

函数的参数在调用时并不要求必须设置，可以不传参数或者省略部分参数，未设置的参数的默认值为 `undefined`，实例代码如下：


```
function sum(a, b, c) {
  a = a || 0;           // 为参数设置默认值
  b = b || 0;
  c = c || 0;
  return a + b + c;
}
var result = sum(1, 2);
```

提示: 因为参数并不强制要求在调用时传递, 所以在函数内对参数的检查和设置默认值是很有必要的, 避免因调用者未按照预想设置参数, 引起异常或产生错误结果。

- arguments

函数体内也可以通过一个隐形的变量 `arguments` 来获取所有参数。`arguments` 是一个类似于数组的对象, 可以通过 `arguments.length` 来获取实际设置的参数数量, 通过 `arguments[0]`、`arguments[1]`、`arguments[n]` 来引用指定位置参数, 实例代码如下:

```
function add() {
  if (arguments.length <= 0) return;
  var m = arguments[0] || 0;           // 为参数设置默认值
  var n = arguments[1] || 0;
  return m + n;
}
```

提示: `arguments` 并不是一个数组, 但如果想像使用数组的方法一样对 `arguments` 进行操作, 可以通过 “`Array.prototype.[xxx].apply`” 方法来达到目的。

4. 局部变量

函数体内声明的变量为局部变量。局部变量在函数体内部使用, 函数外部无法访问。实例代码如下:

```
var total = 0;
function sum() {
  var n = arguments[0] || 0;
  total += n;
  return total;
}
sum(1);
console.log(total);           // 结果为1
console.log(n);               // 异常, n 未定义
```

上面的例子中, 代码最后一行引用了变量 `n`, 因为函数 `sum` 中声明的变量 `n` 是局部变量, 外

部无法访问，所以这里会引发变量未定义异常。可以利用局部变量的这个特点，把代码用函数进行封装，使变量局部化，避免与其他的全局变量冲突。

5.2 事件

5.2.1 事件概述

事件 (Event) 是一种异步编程的实现方式，是程序各个组成部分间的通信。例如，当用户单击页面上的一个元素时，浏览器就会将该动作通过 Click 事件告诉应用程序。事件不是 JavaScript 对象，只是一种传递信息的机制，所以事件本身不能承载任何数据内容。浏览器通过 JavaScript 的 Event 对象来承载事件数据信息。当事件发生时，浏览器将被触发元素、发生位置等相关的原始数据存入 Event 对象，然后程序通过事件监听获取响应数据。

要在 JavaScript 编程中使用事件，首先要了解一些基本概念。

- 事件类型 (Event Type)：用来描述发生的事件类别，即事件的名称。比如，`keydown` 表示按下某个键盘按键，`mousedown` 表示按下鼠标按键。
- 事件目标 (Event Target)：指发生事件的对象。事件触发后都会有一个相应的发生源，这个“源”就是事件的目标。
- 事件处理程序 (Event Handler)：处理所发生事件的程序代码。当在特定的目标上发生特定事件时，浏览器会调用监听该事件的处理程序。
- 事件对象 (Event Object)：包含事件详细信息的 JavaScript 对象。当事件发生后，事件相关的信息传递至监听函数，事件对象即数据信息的载体。
- 事件传播 (Event Propagation)：事件发生时，由 `window` 节点发出，不断经过子元素到达目标元素，然后事件会原路返回，直至回到 `window`。

早期的 Web 标准中，只提供了少数的事件，比如 `load`、`click`、`mouseover` 事件等。随着互联网的快速发展，特别是 HTML 5 的逐渐普及，浏览器所支持的事件集合快速壮大，以至于很难通过单个标准来定义这些事件。下面从事件类型的角度，简单归纳一下 Web 浏览器中常用的事件，见表 5.2。

表 5.2 Web 浏览器常用事件

事件类型	事件名称	事件说明
浏览器窗口事件	<code>load</code>	页面加载完成时触发
	<code>beforeunload</code>	窗口关闭之前触发

续表

事件类型	事件名称	事件说明
	unload	窗口关闭时触发
	focus	窗口得到焦点时触发
	blur	窗口失去焦点时触发
	error	页面上有脚本报错时触发
	resize	窗口大小改变时触发
	contextmenu	弹出右键菜单时触发
鼠标事件	mousedown	当在元素上按下鼠标按钮时触发
	mouseover	当鼠标指针移动到元素上时触发
	mousemove	当鼠标指针在元素上移动时触发
	mouseout	当鼠标指针移出元素时触发
	mouseup	当在元素上释放鼠标按钮时触发
	mousewheel	当在元素上滚动鼠标滚轮时触发
键盘事件	keydown	当用户按下按键时触发
	keypress	当用户敲击按键时触发（晚于keydown）
	keyup	当用户释放按键时触发
表单事件	focus	当表单元素获取焦点时触发
	blur	当表单元素失去焦点时触发
	change	当表单元素的值被改变时触发
	input	当表单元素获得用户输入时触发
	select	当元素内容被选中时触发
	submit	当提交表单时触发
拖放事件	drag	当元素被拖动时触发
	dragstart	在拖动操作开始触发
	dragover	当元素在有效拖放目标上正在被拖动时触发
	dragenter	当元素已被拖动到目标区域时触发
	dragleave	当元素离开有效目标时触发
	dragend	在拖动操作末端触发
	drop	当被拖元素放置到目标区域时触发

注意：以上只列举了部分 JavaScript 事件，此外还有很多不常用的事件和 HTML 5 新增事件，如音频、视频、打印等，要了解更多事件集合，请参阅 W3C 官方文档 <https://www.w3.org/TR/2002/WD-DOM-Level-3-Events-20020208/events.html>。

5.2.2 事件委托

通过上一节学习已经了解到，浏览器为开发者提供了诸多事件类型。通过这些事件类型，使 Web 应用程序能够感知用户的行为和浏览器操作的变化。本节将带领大家学习如何正确使用事件。

事件模型随着互联网的发展而不断演变，早期的事件模型通过 DOM 元素属性实现，即直接以对象属性的形式为 DOM 元素注册事件，称之为 DOM 事件模型，实例代码如下：

```
document.querySelector("#div1").onclick = function(ev){};
```

上述代码的意思就是给 ID 为“div1”的元素注册一个 Click 事件。另外一种写法也表示相同的含义，代码如下：

```
document.querySelector("#div1")["onclick"] = function(ev){};
```

这两句代码虽然含义相同，但在具体开发中第 2 种方式更为灵活，可以动态地注册事件类型。不过，不管选择哪种事件注册方式，都有一个不可避免的缺陷，即后注册的事件会覆盖先注册的事件，看以下这段代码：

```
document.querySelector('#div1').onclick = function (ev) { alert('Hello'); };
document.querySelector('#div1').onclick = function (ev) { alert('World'); };
```

开发者的本意是想在单击 ID 为“div1”的元素时，先弹出“Hello”提示，再弹出“World”提示，但实际上浏览器只会弹出“World”。原因是，后添加的事件监听取代了先前事件。为了解决这种尴尬的问题，DOM2 事件模型应运而生。DOM2 事件模型主要实现了以下两点技术：

- 支持为同一 DOM 元素注册多个同类型事件。
- 把事件分为捕获阶段和冒泡阶段。

基于 DOM2 事件模型的事件监听实现，代码如下：

```
document.querySelector('#div1').addEventListener('click', function (ev) {}, false);
```

通过元素对象的 `addEventListener` 方法为其添加事件监听。使用该方式，在多次监听事件时，不会像之前那样彼此覆盖，每一个监听均有效。`addEventListener` 方法接受 3 个参数。

- **type**：必选，String 类型，指明事件类型（也叫事件名称）。
- **listener**：必选，EventListener 类型，事件处理程序，可以是 JavaScript 方法，也可以是一个匿名方法体。
- **useCapture**：可选，Boolean 类型，指定事件是否发生在捕获阶段。默认值为 `false`，表示事件发生在冒泡阶段。

事件模型中有两种机制。

- **事件捕获 (Capture Phase)**：是指一个事件发生后，从 Window 发出，不断经过下级节点，直到目标节点。在事件到达目标节点之前的过程，就是捕获阶段。而所有经过的节点，都会触发对应事件。

- 事件冒泡（Bubbling Phase）：当事件到达目标节点之后，会沿着捕获阶段的路线返回，因为类似水泡浮起，故称作“冒泡”。通过冒泡这个事件机制，所有子节点的事件都会被其父级节点所捕获。

对于事件的捕获阶段和冒泡阶段，如图 5.1 所示很形象地做出了解释。

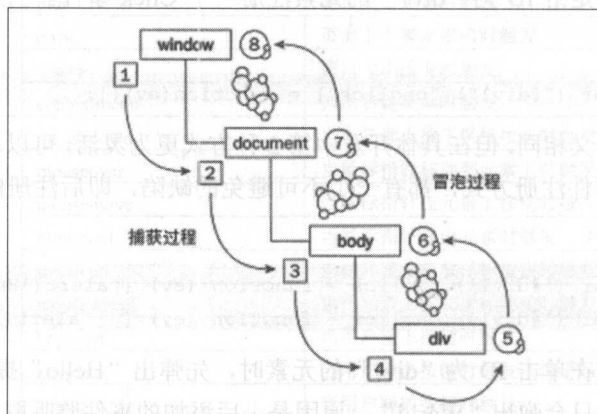


图 5.1 事件的捕获过程和冒泡过程示意图

当事件被触发时，事件监听函数接收一个 Event 对象，即事件对象（Event Object）。Event 对象包含了事件发生时相关的详细信息，比如坐标、元素等。配合事件冒泡的机制，可以在 DOM 根元素上监听所有元素的事件，实例代码如下：

```
<body>
  <a>link</a>
  <b>bold</b>
  <i>italic</i>
</body>
<script type="text/javascript">
function showMe(ev) {
  alert(ev.target.tagName);
}
document.querySelector('body').addEventListener('click', showMe, false);
</script>
```

上述代码首先监听 Body 元素的 Click 事件。单击 Body 元素内的 a、b、i 元素，可弹出目标节点的标签名称。这种将事件交给父元素或祖先元素处理的方式，就叫事件委托（Event Delegation）。

事件委托主要有以下两个优点。

- 提高性能：每一个函数都会占用内存空间，只添加一个事件处理程序，所占用的内存空间更少。
- 动态监听：使用委托可以监听“未来”的元素。比如上面代码中，将 Click 事件委托给元素 Body 处理，那么不管 Body 的子元素后续会新增多少和如何发生变化，只要单击该元素都会触发 Body 元素的监听函数，比如实例代码中的 showMe 函数。

提示：在实际开发中，可能需要移除事件监听，在 DOM0 中只需要将 onclick 设置成 null 即可，而 DOM2 中移除事件监听是通过方法 removeEventListener 完成的，有关该方法的详细介绍，请参阅 http://www.w3schools.com/jsref/met_element_removeeventlistener.asp。

5.2.3 移动端事件

伴随智能手机的普及，专门为移动端设备设计的事件也随之而来。这些事件主要包含三类：触摸事件、手势事件和传感器事件。通过这些特有事件，Web 程序可以轻松获取用户行为，进而做出响应。

1. 触摸事件

触摸事件（Touch Event）是最常见、也是使用范围最广的移动端事件，包含以下四类。

- touchstart：当手指触摸屏幕时触发。
- touchmove：当手指在屏幕上滑动时触发。
- touchend：当手指离开屏幕时触发。
- touchcancel：当系统停止跟踪触摸时触发。

触摸事件是一个从触摸到移动，然后离开的过程，这跟 PC 端的鼠标事件非常类似，如同 mousedown、mousemove 和 mouseup，表示鼠标的按下、移动和释放。

上一节介绍事件发生时，会生成一个 Event 对象，事件相关的信息都会被存储在这个 Event 对象中。移动端事件采用相同的机制，通过 Event 对象传递信息。如触摸事件，Event 对象是 TouchEvent 类型，除了包含一般事件都有的信息外，还包括如下信息。

- touches：表示当前跟踪的触摸操作 touch 对象的集合。
- targetTouches：当前事件目标上 touch 对象的集合。
- changeTouches：表示至上次触摸发生了改变的 touch 对象的集合。

每个 touch 对象同样包含相关事件信息如下。

- **clientX**: 触摸目标在视口中的 X 坐标。
- **clientY**: 触摸目标在视口中的 Y 坐标。
- **pageX**: 触摸目标在页面中的 X 坐标。
- **pageY**: 触摸目标在页面中的 Y 坐标。
- **screenX**: 触摸目标在屏幕中的 X 坐标。
- **screenY**: 触摸目标在屏幕中的 Y 坐标。
- **target**: 触摸的 DOM 节点。

通过一个实例演示移动端的触摸事件，代码如下：

```
01 var touch0 = null;
02 document.body.addEventListener('touchstart', function (ev) {
03     touch0 = ev.touches[0];
04     console.log('touchstart');
05 });
06 document.body.addEventListener('touchmove', function (ev) {
07     var touch = ev.touches[0];
08     var moveX = touch.clientX - touch0.clientX;
09     var moveY = touch.clientY - touch0.clientY;
10     console.log('move:', moveX, moveY);
11 });
```

实例中当手指在屏幕上滑动时，控制台输出滑动的距离，运行结果如图 5.2 所示。

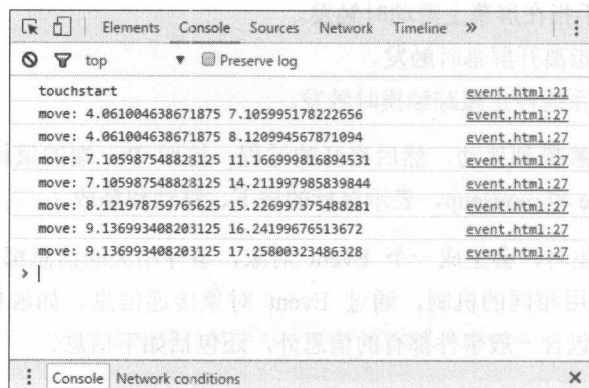


图 5.2 触摸事件运行结果

2. 手势事件

(Gesture Event) 是在开发中使用较少的一类移动端事件，是对触摸事件的高级封装，包含以

下三种类型。

- **gesturestart**: 当手势开始（有 2 根或多根手指触摸屏幕）时触发。
- **gesturechange**: 当手势改变（有 2 根或多根手指触摸屏幕，且发生移动）时触发。
- **gestureend**: 当手势结束（倒数第 2 根手指离开）时触发。

手势事件的 Event 对象是 **GestureEvent** 类型，除了包含基础的 **screenX**、**screenY**、**clientX**、**clientY** 等信息外，同时还包括如下信息。

- **scale**: 缩放比例，即手指移动过程中分开/合拢的比例。默认值为 1，手指相对初始位置分散开来，**scale** 值增大；手指相对初始位置合拢缩短，**scale** 值随之减小。
- **rotation**: 旋转角度，也就是手指间连线旋转的角度。默认值为 0.0，逆时针旋转为负值，顺时针旋转为正值。

3. 传感器事件

传感器事件（Sensor Event）一般用于判断设备在三维空间的位置，图 5.3 清晰地标明了移动设备在三维空间的坐标走向。

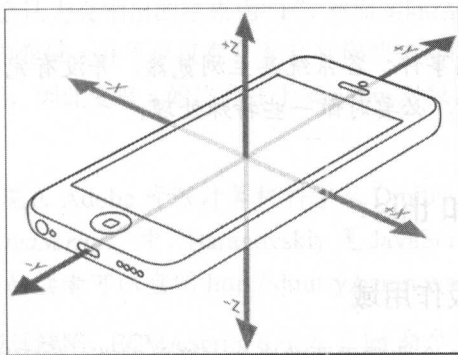


图 5.3 移动设备在三维空间的坐标示意图

常用的传感器事件有如下三种。

- **deviceorientation**: 提供设备的物理方向信息，表示为一系列本地坐标系的旋角。
- **devicemotion**: 提供设备的加速信息，表示为定义在设备上的坐标系中的笛卡儿坐标。另外还提供了设备在坐标系中的自转速率。
- **orientationchange**: 提供设备的旋转方向信息，标明设备当前处于垂直或水平状态。

deviceorientation 事件的 Event 对象为 **DeviceOrientationEvent** 类型，主要包含以下三种信息。

- **alpha**: 在围绕 Z 轴旋转时（左右旋转），Y 轴的度数差。
- **beta**: 在围绕 X 轴旋转时（前后旋转），Z 轴的度数差。
- **gamma**: 在围绕 Y 轴旋转时（扭转设备），X 轴的度数差。

如果将设备放置在水平表面上，屏幕顶端指向西方，则其方向信息值为：

```
{ alpha: 90, beta: 0, gamma: 0 }
```

devicemotion 事件的 Event 对象为 DeviceMotionEvent 类型，包含以下四种信息：

- **acceleration**: 设备的加速度，包含 X 轴、Y 轴和 Z 轴上的 3 个值。
- **accelerationIncludingGravity**: 设备的加速度，考虑重力加速度，同样包括 3 个值。
- **interval**: 事件通知的时间间隔，单位为毫秒。
- **rotationRate**: 设备旋转的速度，数据包含 alpha、beta 和 gamma。

orientationchange 事件监听设备的方向变化，不过稍有不同，Event 对象并未包含方向信息，设备当前的方向通过 screen.orientation.angle 对象获取。其取值 0 和 180 表示竖屏，90 和 270 表示横屏。而 iOS 系统中通过 window.orientation 来获取设备方向，0 和 180 表示竖屏，90 和 -90 表示横屏。

注意：对于个别移动端事件，各系统甚至浏览器，并没有完全统一标准，具体开发中还是需要区别对待，必要时做一些特殊处理。

5.3 作用域、闭包和 this

5.3.1 使用 let 实现块级作用域

作用域是程序设计领域一个非常重要的概念。简单来说，作用域就是变量与函数的可访问范围，即控制着变量与函数的可见性与生命周期。在 ECMAScript 6 之前，JavaScript 只有两种作用域，即全局作用域和函数作用域，实例代码如下：

```
var scope = "global";
function foo() {
  var scope = "local";
  console.log(scope);
}
foo(); // 输入结果: local
console.log(scope); // 输入结果: global
```

在很多流行语言中还存在块级作用域 (Block Scope)，指在一个代码块中所定义的变量，代码块一般以花括号包裹，在这之中定义的变量在代码块之外不可见。而在 JavaScript 中，尽管从语法层面看上去似乎是支持这种块级作用域的，但实际情况却是在 ECMAScript 6 之前的 JavaScript 中并不存在这种作用域。实例代码如下：

```
for(var scope = 0; scope < 5; scope++){
  console.log(scope);
} // 输入结果: 5
```

上述的代码中，变量 `scope` 被定义在 `for` 循环语句的代码块之内，在循环之外引用 `scope` 变量由于缺少了块级作用域，变量被扩散到全局作用域中，因此可以被正常输出。虽然在上面的代码中并未产生多大的危害，但这种混乱却很有可能成为错误的根源。缺少块级作用域不仅会导致代码块执行完毕后变量不会被回收，并且由于定义在代码块中的变量被扩散到上一层作用域中，还会导致另外一种称之为变量声明提升的副作用，接着来看下面的代码：

```
if (!("a" in window)) { var a = 1; }
console.log(a); // 这里输出什么？
```

这段代码似乎是表示“如果 `window` 中不存在 `a` 属性，就声明一个变量 `a`，并将 `a` 赋值为 1，输出这个 `a` 的值”。读者也许会认为会输出的结果为“1”，然而实际结果却是输出了“`undefined`”。之所以出现这种情况，是因为变量 `a` 并不仅仅存在于 `if` 代码块中，由于不存在块级作用域，变量 `a` 的声明被前置于全局作用域，因此变量 `a` 始终存在于 `window` 对象，`if` 语句内的赋值并不会被执行。

注意：这段代码最早出现在 Adobe 资深计算机科学家 Dmitry Baranovskiy 的文章“*So, you think you know JavaScript?*”中，Baranovskiy 是 JavaScript 矢量图形库 Raphaël 的作者之一。有兴趣的读者可以查阅 <http://dmitry.baranovskiy.com/post/91403200>。

由于 JavaScript 的这种设计缺陷，ECMAScript 6 引入了 `let` 命令来改变缺少块级作用域导致混乱的问题。`let` 命令与 `var` 命令同样用来声明变量，而与 `var` 不同的是，`let` 声明的变量只存在于其所在的代码块中，实例代码如下：

```
function foo() {
  let scope = "function";
  if (true) {
    let scope = "block";
    console.log(scope); // 输入结果: block
  }
  console.log(scope); // 输入结果: function
}
```

在 ECMAScript 6 块级作用域出现之前,在多人协作的项目中,为了避免由于定义相同的变量名而产生的异常,通常会使用匿名函数划分各自的作用域。而在上述代码中,由 `let` 定义的变量只存在于各自的作用域中,并不会发生重复定义同名变量导致被覆盖的错误。修改先前 `for` 循环实例,改用 `let` 定义变量 `scope`,在代码块外引用 `scope` 变量抛出异常错误,代码如下:

```
for(let scope = 0; scope < 5; scope++){
  console.log(scope); // ReferenceError
```

同样,使用 `let` 命令定义的变量,并不会产生声明前置的效果,即表示变量必须在声明后才能使用,未声明前的引用会导致错误,这在语法上称之为“暂时性死区”(Temporal Dead Zone,简写 TDZ)。实例代码如下:

```
typeof a; // undefined
var a = 1;
typeof b; // ReferenceError
let b = 1;
```

注意: 本节所介绍的声明提升现象也存在于函数,而 ECMAScript 6 规定函数本身的作用域在其块级作用域之内,这使得函数声明与 `let` 命令产生了类似效果。

5.3.2 闭包

上一节介绍了 JavaScript 作用域的概念,并且了解了 JavaScript 函数可以用来创建作用域。JavaScript 查找变量即变量解析的过程,首先在当前定义的局部作用域中查找,如果未发现,就会查找上一层作用域。由于 JavaScript 是基于词法(静态)作用域的语言,词法作用域的含义是在函数定义时就确定了作用域,而不是函数执行时再确定。先来看一个例子,代码如下:

```
function outer() {
  var scope = 10;
  return function inner() {
    scope += 10;
    console.log(scope);
  }
}
var scope = 100;
var fn = outer();
fn(); // 输出显示: 20
```

上述实例输出结果为“20”而不是“110”,这是由于变量 `scope` 被定义为 `outer` 函数的局部变量,而在 `inner` 函数定义时即确定了其作用域,并引用了其外部函数 `outer` 的局部变量 `scope`。

函数作为 `outer` 函数的执行结果被返回，当 `outer` 函数在执行完毕后，定义在其内部的变量 `scope` 并没有被回收，而是可以通过函数 `fn` 的执行被获取，这里的 `inner` 函数，即形成了闭包。

注意：来自维基的定义，在计算机科学中，闭包 (Closure) 是词法闭包 (Lexical Closure) 的简称，是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。可以参考来源 [https://en.wikipedia.org/wiki/Closure_\(computer_programming\)](https://en.wikipedia.org/wiki/Closure_(computer_programming))。

不少开发者常常搞不清楚匿名函数与闭包的关系，也有认为函数即是闭包。虽然从技术角度，不能说这种看法一定是错误的。但从严格意义上来说，构成闭包需要有对上下文词法作用域中变量的引用，并在外部函数执行完毕时，被引用的变量并不会被垃圾回收器回收。听起来还是有点绕，不妨将上述实例的 `fn` 函数再次执行，这时会发现返回值从“20”变成了“30”。在理解了闭包的运行机制后，紧接着再看一个例子，代码如下：

```
var scope = 10;
function calculate(addend){ console.log(scope + addend); }
(function(ca){
    var scope = 100;
    ca(5);
    // 输出显示: 15
})(calculate);
```

在 JavaScript 中，函数是“第一类对象” (First-class Object)，这让 JavaScript 函数可以被存入变量或其他结构，也可以被作为其他函数的返回值，或者被作为参数传递给其他函数。像上述实例，当 `calculate` 函数作为参数传递给自执行函数时，同样出现了闭包的身影。所以本质上，将函数作为其他函数的返回值和作为参数传递时，常常就会不经意间创建闭包，如在使用定时器、事件监听亦或是 AJAX 回调函数时。前面的两个例子也分别诠释了这两种情况。

注意：第一类对象 (First-class Object) 在 1960 年由 Christopher Strachey 发明，原来称之为第一类公民 (First-class citizen)，意思是指函数可以作为电脑中的第一类公民。英文中也称之为 First-class entity 或 First-class value。

了解了闭包的原理，不妨来看看闭包的一些使用场景。上面的例子已经介绍了闭包导致内存未释放的情况，这让闭包可以用来保存中间计算结果，类似于实现了计算结果的缓存。下面将使用闭包函数计算斐波那契数列，实例代码如下：

```
var count = 0,
fib = (function(){
    var arr = [0,1,1];
    // 前 3 位直接返回
```



```

return function(n){
    count++;
    var res = arr[n];                // arr 一直在内存中
    if(res){
        return res;
    }else{
        arr[n] = fib(n-1) + fib(n-2);
        return arr[n];
    }
}
})();
fib(10);                            // 结果: 55

```

闭包另一个使用场景是实现内部变量的封装, 即使用匿名函数封装私有成员的单例模式, 也称作模块模式。该模式最早由 Yahoo 的工程师提出, 并广泛运用在早期的前端框架 YUI 中, 简单模仿该方式的实现, 代码如下:

```

YAHOO.Util = function () {
    var privateVar = "";            // 私有属性只能内部访问
    var privateMethod = function () { // 私有方法只能内部访问
        // TODO
    }
    return {
        PublicProperty: "",        // 暴露对外的公共属性
        PublicMethod: function () { // 暴露对外的公共方法可以访问内部属性和方法
            // TODO
        }
    };
}();

```

注意: 关于模块模式, 可以在 Yahoo 的技术博客中了解更加详细的说明, 文章地址为 <http://yuiblog.com/blog/2007/06/12/module-pattern/>。

说明: 滥用闭包也会导致一些诸如内存泄漏的性能问题, 这点在低版本的 Internet Explorer 浏览器中表现得尤为明显, 合理使用好闭包是成为一个优秀的前端工程师的必备要求。

5.3.3 采用 call、apply、bind 改变 this

JavaScript 的 this 关键字表示函数运行时生成的内部对象。和变量的搜索过程不同, this 的值从执行上下文中获取, 而不会在作用域链中搜寻。在面向对象的程序设计语言中, this 关键字指代

当前对象，通常在编译期确定，称为编译期绑定，而在 JavaScript 中，`this` 是动态绑定的，称为运行期绑定。`this` 可以是全局对象、当前对象或者任意对象，这完全取决于函数的调用方式，不同的调用方式会产生不同的执行上下文，也就会有不一样的 `this` 含义。

在 JavaScript 中，函数的调用方式有以下几种：作为函数调用、作为对象方法调用、作为构造函数调用以及使用 `call` 和 `apply` 调用。首先来看一个简单实例，代码如下：

```
var name = "tiger";
function Animal () {
    var name = "cat";
    console.log(this.name);
}
Animal(); // 输出结果: tiger
```

上述代码中，`this` 并不在函数作用域中查找“`name`”对象，`Animal` 函数在全局上下文中执行，因此 `Animal` 内部的 `this` 指向全局对象。该全局对象在浏览器环境中即 `window` 对象，在全局作用域下使用 `var` 定义的变量会定义在 `window` 对象下，因此执行 `Animal` 函数得到 `this.name` 的值为“`tiger`”。接下来，修改一下实例代码，使用 `call` 调用函数，代码如下：

```
var name = "tiger";
function Animal () {
    console.log(this.name);
}
Animal.call({name: "lion"}); // 输出结果: lion
```

当使用 `call` 调用函数时，`this` 被指向传入的对象，`this.name` 的值为“`lion`”，使用 `apply` 也可以达到相同的效果。ECMAScript 5 新增加的 `bind` 方法，在实现上同 `call` 和 `apply` 稍有不同，调用 `bind` 方法会返回一个新的函数，称为绑定函数。当调用绑定函数时，会以创建时所传入的第一个参数作为 `this` 指向的对象。修改上述的 `Animal` 函数使用 `bind` 方法改变 `this` 指向，代码如下：

```
Animal.bind({name: 'lion'})();
```

由于 `call`、`apply`、`bind` 可以改变执行上下文的特性，所以开发者可以使用基础类型对象内置的原型方法。比如，调用 `Object` 原型对象上的 `toString` 方法判断对象类型，代码如下：

```
Object.prototype.toString.call([]) === "[object Array]"; // 结果: true
```

类数组对象使用数组方法，代码如下：

```
function Add() {
    return [].slice.call(arguments).reduce((a, b) => a + b);
}
```

```
Add(1, 3, 5, 7, 9);
```

```
// 返回结果: 25
```

注意：call、apply 和 bind 方法的第一个参数都是 this 指向的对象，在“非严格模式”下，如果传入的对象是 null 或者 undefined，则 this 指向全局对象。call 和 apply 在实现效果上并没有什么不同，只是参数格式略有差异。

5.4 面向对象

面向对象是程序设计语言中一种常见的思想。就现实世界而言，整个世界是由各种各样具备规律和状态的对象所组成，这是一种认知自然世界的哲学。JavaScript 是一种基于对象，但书写上又不同于传统面向对象编程的一门语言。本节将向读者介绍 JavaScript 实现面向对象编程的几种方式（包含了 ECMAScript 6 语法）。

5.4.1 原型和原型链

JavaScript 不同于大多数面向对象语言，其函数没有签名，所以并不支持接口继承。在 JavaScript 中继承由原型链（Prototype Chain）来实现，对象原型（Prototype）的概念常常会困惑许多 JavaScript 的初学者，而事实上基于原型的继承模型比传统的类继承更要强大，JavaScript 可以通过原型的继承来模拟类继承，而一个类继承的模型想要模拟原型继承却要难很多。

在 JavaScript 中每个对象都有一个指向其原型对象的内部引用，而这个原型对象又通过引用指向其原型，直到某个对象的原型为 null，这种一级一级的链结构就被称为原型链。

1. 继承属性和方法

这种基于原型的继承第一个表现在对象的继承属性（方法也包含其中），JavaScript 对象有一个指向原型对象的链，当访问一个对象的属性时，JavaScript 不仅会在该对象上查找，也会在该对象的原型上查找。

继承方法需要注意的是，当方法被调用时的 this 指向为当前继承的对象，而不是继承的函数所在的原型对象，实例代码如下：

```
var o = { // 一个基本对象包含两个属性，其中一个属性是方法
  a: 2,
  b: function() { return this.a + 1; }
}
console.log(o.b()) // 当调用 o.m 时，this 指向是 o，所以 o.b() 为 3
var p = Object.create(o);
```

```
p.b = 12 // 创建 p 的属性 a，当原型链在 p 中查找 a 时，先找这个值
console.log(p.b()) // 当调用 p.m 时，this 指向的是 p，所以 o.b() 为 13
```

2. 创建对象和生成原型链

创建对象和生成原型链的方法主要有四种（ECMAScript6 增加了 class 关键字），本节主要介绍前三种，如下。

（1）使用普通语法创建对象

使用普通语法创建对象即一般的 JavaScript 创建变量的方法，代码如下：

```
var o = {a: 1};
```

（2）使用构造器创建对象

构造器是一个普通的函数，当使用 new 操作符来给这个函数创建实例时，这个函数就称为构造器。代码如下：

```
function Person (name){
    this.name = name || "无名氏";
};
var p = new Person ("小张");
```

（3）使用 Object.create 创建对象

该方法在 ECMAScript 5 中被加入，可以方便地通过一个对象原型创建一个新的对象，新对象的原型即使用 create 方法传入的第一个参数。语法如下：

```
Object.create(prototype, descriptors)
```

3. __proto__、prototype 和 constructor

“__proto__”属性是 ECMAScript 标准在 JavaScript 上的实现，这个属性相当于“someObject.[[Prototype]]”，用来指派 someObject 的原型，存在于所有对象之上。比如存在一个对象“var a = { b: 1}”，那么 a 的“__proto__”就是一个空的对象。而一个字符串的“__proto__”是一个 String 对象，String 对象的“__proto__”又再指向空对象。

prototype 则是构造函数拥有的一个属性，这个属性指向的是构造函数的原型对象，原型对象的存在意味着这个对象的所有属性和方法，都可以被构造函数的实例继承。而对开发者而言就需要把所有的共享属性和方法放入这个对象中，在实例化中只需要引用这部分即可，减少了对内存的占用。

那么“__proto__”和 prototype 的关系是什么呢？拿一个简单的 string 来说，这个 string 的

“__proto__”如前文所述是一个名为 String 的对象，这个对象就是 String 构造器的原型对象，而 String 构造器的 prototype 就是 String 实例的“__proto__”。实例中一个构造器创建对象，代码如下：

```
var o = new Foo(); // 构造器创建对象
```

构造器创建对象的过程可以理解为：

```
var o = new Object(); // 用 Object 构造器构造一个对象
o.__proto__ = Foo.prototype; // 将对象的__proto__指向构造器的prototype
Foo.call(o); // 初始化 o
```

现在可以通过“__proto__”属性找到实例的原型对象，那么如果需要寻找实例的构造函数，就要使用 constructor 属性，这个属性会指向实例的构造函数，假设存在一个实例对象 o，代码如下：

```
o.__proto__ = o.constructor.prototype;
```

对于开发者来说，prototype 是一个最常用的属性，这个属性能帮助开发者很好地实现继承，但实际使用中，仍会存在一些多继承的需求或者对于一些类语言的开发人员更希望使用类的结构来实现继承，这些问题会在接下来的章节进行解释。

5.4.2 Mixin 模式

JavaScript 通过原型链实现继承关系，但该继承基于单一的原型链，单一原型链意味着只能继承唯一原型，如果一个构造函数需要另外原型上的属性，只能在该构造函数的原型上再实现一遍。

在介绍 Mixin 之前，思考一个问题：如果需要继承多个结构怎么办？在 C++ 中引入了一个多重继承的概念，多重继承虽然解决了这个问题，但是会给程序造成许多不必要的复杂度。在 Java 中设计了接口继承来实现多重继承，对于 JavaScript 这样一个没有类概念的脚本语言，实际在语言层面上并没有考虑多重继承。Mixin 模式是开发者引入解决多重继承的方案。Mixin 实现多重继承简单地说是—种组合，因为 JavaScript 继承来自原型链，所以需要做的就是将多个继承对象上的属性拷贝到一个原型上，一个简单的 Mixin 模式代码实现如下：

```
// mixin 方法，接收一个构造函数和一个原型对象
var mixin = function (obj, mixins) {
  var newObj = obj; // 保存构造函数等待扩展原型链
  newObj.prototype = Object.create(obj.prototype); // 原型对象实例赋给新的函数的原型
  for (var key in mixins) { // 遍历原型对象上的属性
    if (mixins.hasOwnProperty(key)) { // 如果是对象自身属性
      newObj.prototype[key] = mixins[key]; // 复制给新函数的原型
    }
  }
}
```

```

    }
  }
  return newObj; // 返回新的构造函数
}
var mixinObj = { // 要混入的原型对象
  foo: function() { // 一个属性为函数 foo
    console.log('foo');
  }
}
var MyFunc = function() { console.log('bar') } // 构造函数
var NewFunc = mixin(MyFunc, mixinObj); // mixin 出一个新的构造函数
var newFunc = new NewFunc; // 实例化
newFunc.foo(); // 输出了mixin上的foo

```

上述实现方法存在一个问题，如果两个继承对象上拥有同名属性，只可以保留一个，具体保留哪一个取决于 Mixin 方法的实现。上述的代码会选用被 Mixin 对象的方法的属性覆盖前一个。

对于一些现代的 Web 框架来说，实例实现的 Mixin 方法能力不足，像 React、Vue.js 会对其中的组件生命周期的属性进行合并执行，代码的实现取决于框架本身，但对于开发者来说，如果需要自行实现一个 Mixin 方法，可以考虑一些具体属性的合并执行。Mixin 模式的实现从类的角度来看并不优雅，在 ECMAScript 7 中提供了装饰器语法。装饰器可以对类进行修改，下面提供一个使用装饰器实现 Mixin 模式的实例，代码如下：

```

function handleClass() { // 处理类
  for (let i = 0, l = mixins.length; i < l; i++) { // 遍历混入的对象
    const descs = getOwnPropertyDescriptors(mixins[i]) // 获取 mixins 的属性
    for (const key in descs) { // 遍历原型对象上的属性
      if (!(key in target.prototype)) { // 如果类上没有这个属性
        defineProperty(target.prototype, key, descs[key]); // 定义一个属性
      }
    }
  }
}
export default function mixin(...mixins) { // 输出这个方法
  return target => { // 返回一个对类包装的函数
    return handleClass(target, mixins);
  }
}

```

通过以下方式调用，代码如下：

```
@mixin(mixinObj)
```

```
class A {};
```

上述实例中把 `mixinObj` 混入 `class A` 中，实际上装饰器可以理解为对 `class A` 的一层函数包装，等同于“`mixin(mixinObj)(A)`”。开发者也可以在模块输出时进行一层类似的函数包装来实现 `Mixin`，这种模式在一些框架中（比如 `React`）被称为高阶组件。

5.4.3 ECMAScript 6 的 Class 和 Extends

JavaScript 在 ECMAScript 6 中引入了 `Class` 和 `Extends` 关键字来实现类和类的继承。尽管原型链已经可以很好地实现继承，但是对于传统面向对象语言的开发者，原型链是很容易让人产生困惑的，`Class` 和 `Extends` 作为语法糖解决了开发者对于 JavaScript 继承的困惑。

类的形式代码如下：

```
class People {                                // 定义一个类 People
  constructor(name) {                        // constructor 函数，必须存在，接收实例化参数
    this.name = name;
  }
  getName() {                                // 类的属性
    console.log(this.name)
  }
}
var p = new People('John');                  // 实例化一个 Person 类
p.getName()                                  // 调用实例的 getName 方法，输出 'John'
```

JavaScript 的类包含了一个 `constructor` 方法，这是类的默认方法。当通过 `new` 关键字生成对象实例时，`constructor` 方法自动被调用，构造函数接收的参数即类实例化接收的参数。如果这个方法没有显示声明，会自动生成一个空方法。

另外 `Class` 中可以进行静态声明，这和大多数面向对象语言相同，只需要在属性上加上 `static` 关键字即可，实例代码如下：

```
class People {
  static sayHello() {                        // 定义一个静态方法
    console.log('hello');
  }
}
People.sayHello()                           // 不需要实例化，直接用类调用静态方法
```

`Extends` 关键字配合 `Class` 实现继承，实例代码如下：

```
class Student extends People {              // Student 类继承 People 类
```

```

constructor (name, grade) {          // 声明 constructor 方法
    super(name);                      // 继承父类的 this 对象
    this.grade = grade;
}

getGrade() {                          // Student 类的属性
    console.log(this.grade);
}

}

var s = new Student('Tom', 6);        // 实例化 Student 类
s.getName();                          // 调用继承的属性, 输出 'Tom'
s.getGrade();                         // 调用 Student 类的属性, 输出 6

```

Extends 实现的继承相比原型链的方式更加直观, 子类需要在 constructor 构造函数中调用 super 方法执行父类构造函数。子类所有的 this 调用也需要在 super 执行之后, 否则在类的实例化时会抛出错误, 因为子类实际上并没有自己的 this 对象, 而是通过继承父类 this 对象获取。

5.5 异步编程

在实际项目中, 为了提高用户体验, 通常情况下, 需要采用 AJAX 方式访问接口获取数据, 并在前端渲染页面。本节将从基础的 AJAX 开始, 介绍基于 Callback 方式的异步编程, 同时还将为读者带来 ECMAScript 6 中的 Promise 和 Generator 新方式。

5.5.1 AJAX 中的 Callback 回调函数

AJAX (Asynchronous JavaScript and XML) 基于 XMLHttpRequest 对象实现, 在标准浏览器中使用 AJAX, 实例代码如下:

```

function ajax(url, callback) {
    var xmlRequest = new XMLHttpRequest();          // 新建 XMLHttpRequest 对象
    xmlRequest.onreadystatechange = function() {    // 检测 readystatechange 事件
        if(xmlRequest.readyState === XMLHttpRequest.DONE) { // 当请求已经完成
            if(xmlRequest.status === 200) {          // 如果服务器端响应没有错误
                callback(null, xmlRequest.responseText);
            } else {
                callback(xmlRequest.status);
            }
        }
    };
    xmlRequest.open("GET", url);                    // 建立一个链接
}

```



```
xmlRequest.send(null);
```

```
// 发送 body 为空的数据
```

```
}
```

提示: XMLHttpRequest.DONE 是标准的处理方式, 在 Internet Explorer 平台下, 该属性在不同版本的支持上略有差异。

上述代码中, 通过 XMLHttpRequest 实例化对象的 open 方法建立链接, 并调用 send 方法发送请求。在请求过程中, 该对象的 readyState 属性会随着请求的状态发生改变。通过监听readystatechange 事件, 判断 readyState 属性的值, 来检测请求是否已经完成。当请求完成时, 依据 HTTP 状态码判定请求状态。状态码为 200 表示服务端返回的响应为成功, 采用 Callback (回调函数) 的方式将请求的结果返回给 AJAX 函数的调用方。

XMLHttpRequest 在使用 POST 方式提交数据时, 需要根据服务器端接口实现方式添加对应 HTTP 头。如采用表单方式提交数据, 则需要设置 Content-Type, 代码如下:

```
xmlRequest.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
```

使用先前创建的 ajax 函数异步请求数据, 代码如下:

```
ajax("http://www.hujiang.com", function (err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log('the response body is ' + data);
  }
})
```

5.5.2 Promise 模式

上一节介绍了采用 Callback 方式实现异步编程, 在实际项目中经常发生 Callback 嵌套现象。如先请求了接口 A 的数据, 之后再请求接口 B, 之后再请求接口 C。在这种情况下代码的可读性变得比较差, 代码如下:

```
ajax("http://apiA", function (dataA) {
  ajax("http://apiB", function (dataB) {
    ajax("http://apiC", function (dataC) {
      // ...
    });
  });
});
```

上述代码中, Callback 一层嵌套一层, 层级多了代码的阅读会存在较大障碍。ECMAScript 6

为开发者提供了 **Promise** 对象，有效地解决了这一问题。采用 **Promise** 对象，对上述定义的 **ajax** 函数进行包装，代码如下：

```

01 function ajax(url) {
02     return new Promise(function (resolve, reject) { // 直接返回一个 Promise 对象
03         var xmlRequest = new XMLHttpRequest();
04         xmlRequest.onreadystatechange = function () {
05             if (xmlRequest.readyState === XMLHttpRequest.DONE) {
06                 if (xmlRequest.status === 200) {
07                     // 成功时，结果采用 resolve 回调返回
08                     resolve(xmlRequest.responseText);
09                 } else {
10                     // 失败时，将出错信息采用 reject 回调返回
11                     reject(new Error("请求错误编号:" + xmlRequest.status));
12                 }
13             }
14         };
15         xmlRequest.open("get", url);
16         xmlRequest.send(null);
17     });
18 }

```

上述代码中，创建了一个 **Promise** 对象，该对象的构造参数是一个函数，该函数包含两个参数 **resolve** 和 **reject**。异步调用成功时，其结果通过函数 **resolve** 回调返回；调用失败时，错误信息通过函数 **reject** 回调返回。在调用异步操作时，直接使用 **Promise** 对象的实例方法执行成功和失败操作，代码如下：

```

ajax("http://huiang.com").then(function(data) {
    // data 中包含返回的内容
}).catch(function(err) {
    // err 包含错误异步调用的错误信息
});

```

上述代码中，通过 **Promise** 对象的 **then** 方法，获取异步成功回调的数据，通过 **catch** 方法，捕获异步调用的错误信息，同时 **then** 方法支持多次链式调用，代码如下：

```

ajax("http://someapi").then(function(data) {
    data.a = "some value"; // 修改某些属性
    return data;
}).then(function(data) {
    // some codes
}).catch(function(err) {

```

```
// some codes
});
```

对于多重嵌套的异步调用,可以很方便地通过 **Promise** 实现,只需要在成功回调后,返回一个新的 **Promise** 对象即可,代码如下:

```
ajax("http://api1").then(function(data){
    return ajax("http://api2");           // 返回一个新的 Promise 对象
}).then(function(data){
    // data 参数为 api2 返回的数据
});
```

Promise 对象还提供了一些静态方法,主要如下。

- **Promise.resolve(data)**: 包裹 **Promise** 对象,在 **then** 方法中返回包裹的 **data** 对象。
- **Promise.reject(error)**: 包裹 **Promise** 对象,在 **catch** 方法中返回包裹的 **error** 对象。
- **Promise.all(promises)**: 参数 **promises** 为 **Promise** 对象数组,返回一个 **Promise** 对象。当这个数组中所有的 **Promise** 对象均执行完毕时,该方法返回的 **Promise** 对象才开始执行 **then** 和 **catch** 回调。
- **Promise.race(promises)**: 参数 **promises** 为 **Promise** 对象数组,返回一个 **Promise** 对象。当这个数组中任一 **Promise** 对象执行完成时,该方法返回的 **Promise** 对象开始执行 **then** 和 **catch** 回调。

通过 **Promise** 对象,可以避免出现 **Callback** 多次嵌套后的“复杂”代码。通过 **Promise.all** 和 **Promise.race** 方法,可以容易地控制多个 **Promise** 并行执行。

5.5.3 生成器 Generator

前一节中,介绍了采用 **ECMAScript 6** 的 **Promise** 对象简化多层嵌套 **Callback** 所带来的代码复杂性,通过链式调用,解决了代码可读性问题。本节将介绍 **ECMAScript 6** 中提供的 **Generator** 对象。首先看一个简单的例子,代码如下:

```
function* gen(){           // 定义一个生成器
    yield 1;               // 通过 yield 关键字,产出 1
    yield 2;               // 通过 yield 关键字,产出 2
    return 3;              // 返回 3
}
```

在形式上 **Generator** 生成器是一个函数,通过 **function** 关键字后面增加“*”标记可以看出。在生成器内部,“**yield**”关键字定义多个不同的值。调用生成器和调用普通函数一样,代码如下:

```
var g = gen();
```

但是和函数不同的是，采用上述代码后，生成器并没有执行，返回的不是函数执行的结果，而是返回了一个遍历器对象（Iterator Object）。

提示：Iterator 是 ECMAScript 6 的新语法，遍历器对象可以采用“for...of”循环遍历其所有子项。

下一步，需要调用返回的遍历器对象的 next 方法，依次返回每一项的状态，代码如下：

```
g.next();      // {value: 1, done: false}
g.next();      // {value: 2, done: false}
g.next();      // {value: 3, done: true}
g.next();      // {value: undefined, done: true}
```

在上述的实例中，第一次执行 next 方法时，生成器开始执行到第一个“yield”关键字，并将执行的结果返回。返回的结果为包含“value”和“done”两种属性的对象，属性分别表示值和状态。再次执行 next 方法，将从上次执行的“yield”关键字开始，执行到下一个“yield”关键字停止。当执行最后的 return 返回结果后，生成器代码执行完，其状态属性 done 值变为“done”。如果此时再次调用 next 方法，将返回“{value: undefined, done: true}”结果。

由于生成器的特性，每次调用 next 方法执行一个区间的代码，即生成器提供了分段执行机制。在生成器中使用异步调用，代码如下：

```
function* demo() {
  var a = yield Promise.resolve(1);
  var b = yield Promise.resolve(a + 2);
  var c = yield Promise.resolve(3+b);
  return c;
};
```

上述实例，调用生成器 next 方法，得到的结果中 value 属性为 Promise 对象。在该 Promise 对象完成时，执行生成器的下一个 next 方法，这样即可以实现依次执行每一个 yield 返回的 Promise 对象完成异步调用。代码如下：

```
01 function co(generator) {
02   var gen = generator();
03   function nextFunc(arg) {
04     var next = gen.next(arg);      // 首先执行以下 next，取到第一个 Promise
05     if (!next.done) {              // 如果生成器没有完成时，执行这个 Promise
06       next.value.then(function (data) {
07         console.log(data);
```



```

08         nextFunc(data);           // 当 Promise 执行成功后，执行下一个
09     });
10     } else if(next.value) {
11         console.log(next.value);    // 当生成器执行完成后，得到返回的值。
12     }
13 }
14 nextFunc();
15 }
16 co(demo);

```

上述代码简单地实现了通过生成器实现异步调用。可以发现异步调用方式比直接采用 Promise 调用更为直观，可以像写同步代码一样调用异步代码，代码具有更好的可读性。

提示：实际项目中，需要处理异常以及类型校验，推荐采用开源类库 `co` 实现这一功能，`co` 库的 GitHub 地址为 <https://github.com/tj/co>。

在 ECMAScript 7 中新提供了 `async` 和 `await` 关键字，是一个更加优秀的异步解决方案，代码如下：

```

async function es7(){
    var a = await Promise.resolve(1);
    var b = await Promise.resolve(a * 2);
    var c = await Promise.resolve(b+3);
    return c;
}
es7().then(function(data){           // 异步调用函数执行的结果是 Promise 对象
    console.log(data);
});

```

通过“`async`”关键字，标注函数为异步函数；通过“`await`”关键字，等待异步调用的结果。异步函数执行后，返回结果为 `Promise` 对象，所以从代码上看异步调用和同步调用的代码几乎一致。

5.6 模块化

以功能块为单位进行程序设计，实现其求解算法的方法称为模块化，原则是“高内聚，低耦合”。“高内聚”尽量减少不同文件中函数的交叉引用，“低耦合”是模块与模块之间要相互独立。模块化的目的是为了降低程序复杂度，使程序设计、调试和维护等操作简单化。

5.6.1 为什么需要模块化

在 JavaScript 发展初期, AJAX 并未普及, JavaScript 还只是一种“玩具语言”, 用来在网页上进行表单校验、实现简单的动画效果等。回想一下, 那个网页上到处小广告飘来飘去的时代多么令人痛恶。Web 1.0 相当长一段时间的页面脚本都在以实例中的形式存在着, 代码如下:

```
if(xx){
    //.....
}else{
    //.....
}
for(var i=0; i<10; i++){
    //.....
}
```

代码简单地堆积在一起, 只要能顺利地从上往下依次执行即可。但随着网站越来越复杂, 实现网站功能的 JavaScript 代码也越来越庞大, 网页越来越像桌面程序, 很多问题开始被暴露出来, 比如全局变量冲突、函数命名冲突、依赖关系处理等。实例中 b.js 依赖 a.js, 标签的书写顺序必须按照先后排列, 代码如下:

```
<script type="text/javascript" src="a.js"></script>
<script type="text/javascript" src="b.js"></script>
```

庞大复杂的应用需要一个团队分工协作、进度管理、单元测试等, 开发者不得不使用软件工程的方法管理网页的业务逻辑。

Javascript 模块化编程, 已经成为一个迫切的需求。但是 Javascript 并不是一种模块化编程语言, 不支持“类”(class)和“模块”(module)(ECMAScript 标准第六版支持“类”和“模块”, 但在浏览器环境使用还要借助 Babel 进行编译, 在后面的章节会进行介绍)。经过 JavaScript 开发者和社区的不断努力, 现在已经能够模拟出“模块”的效果, 一起来看下模块化的一些写法的探索。

提示: Babel 是源代码到源代码的编译器, 可以把符合 ECMAScript 6 标准的代码完美地转换成 ECMAScript 5 标准的代码, 并且确保良好地运行在所有主流浏览器中, 官网地址为 <http://babeljs.io/>。

1. 原始写法

既然模块是要实现某个功能, 那么可以把实现功能的一组函数放在同一文件中, 实例代码如下:

```
function a1(){
    // .....
}
function b2(){
    // .....
}
```

函数 `a1` 和 `b2` 组成一个模块，其他文件先加载该模块，再对函数进行调用。这种做法的缺点很明显，一旦代码量庞大后很难保证不发生变量命名冲突，容易“污染”全局变量，而且模块成员之间没有太多必然联系。

2. 添加命名空间

为了解决上面这种写法的缺点，可以加入命名空间来管理模块，也就是使用单全局变量的模式。实例代码如下：

```
var module_speical = {
  _index: 0,
  a1: function(){
    // .....
  },
  b2: function(){
    // .....
  }
}
```

`a1` 和 `b2` 两个函数都封装在对象 `module_speical` 中，调用方法如下：

```
module_speical.a1();
module_speical.b2();
```

通常在属性名前加下画线表示该属性为私有属性，不过这只是一种开发规范上的约定，实际上该属性仍然向外暴露，比如在 `module_speical` 对象上已经添加了 `_index` 属性，可以在外部被改写，代码如下：

```
module_speical._index = 6;
```

怎么才能让私有变量不被暴露呢？来看下面提供的模块化方式。

3. 立即执行的函数

立即执行函数简称“`IIFE`”（Immediately-Invoked Function Expression），实际就是匿名函数，实例代码如下：

```
(function(a){
    console.log(a);
})("hello world")
```

匿名函数能够形成一个独立的作用域，用匿名函数作为一个“容器”，“容器”内部可以访问外部的变量，而外部环境不能访问“容器”内部的变量，所以匿名函数内部定义的变量不会与外部的变量发生冲突。使用立即执行的匿名函数实现模块化，实例代码如下：

```
var module_speical = (function(){
    var _index = 0;
    var a1 = function(){
        // .....
    };
    var b2 = function(){
        // .....
    };
    return {
        a1: a1,
        b2: b2
    }
})();
```

a1 和 b2 两个函数作为匿名函数返回并赋值给变量 module_speical，调用的方法如下：

```
module_speical.a1();
module_speical.b2();
```

这种方式既避免了命名冲突，又使得私有变量 _index 不能被外部访问和修改。如果不使用一些模块化的框架，而只用 jQuery 或者 Zepto 类库完成开发，匿名函数实现模块化是常用的方式。有兴趣的读者可以阅读 jQuery 源码，里面大量采用了这种方式。

5.6.2 AMD 和 CMD 规范

在上一章节中，介绍了模块化的基本概念和常用写法，本节主要介绍如何规范地使用模块化进行开发。时下流行的模块化规范主要有 CommonJS、AMD 和 CMD 规范。比方说，CommonJS 规范的实现代表是 Node.js，AMD 规范的实现代表是 RequireJS，CMD 规范的实现代表是 Sea.js。接着来了解这三种模块化规范。

1. CommonJS 规范

Node.js 是服务器端 JavaScript 解释器，允许开发能够用 JavaScript 的语法去编写服务端程序。Node.js 应用由模块组成，采用 CommonJS 规范，通过全局方法 require 来加载模块，实例代码如下：


```

var http = require('http'); // 引入 http 模块
var server = http.createServer(function(req, res){ // 用 http 模块提供的方法创建一个服务
  res.statusCode = 200; // 返回状态码为 200
  res.setHeader('Content-Type', 'text/plain'); // 指定请求和响应的 HTTP 内容类型
  res.end('Hello World\n'); // 返回的数据
});
server.listen(3000, '127.0.0.1', function(){ // 监听的端口和主机名
  console.log('Server running at http://127.0.0.1:3000'); // 服务启动成功后控制台打印信息
});

```

从上面的实例代码中可以看出, 首先通过 `require` 方法引入 `http` 模块, 接着调用 `http` 模块的 `createServer` 方法创建一个服务, 最后给这个服务指定端口和主机名, 一个简单的 HTTP 服务器就创建好了。这就是 CommonJS 规范模块的实际应用了, 那么如何编写一个 CommonJS 规范的模块呢? 这就需要用到 `Module` 对象。

Node.js 内部提供一个 `Module` 构造函数, 所有模块都是 `Module` 的实例。每个模块内部, 都有一个 `Module` 对象, 代表当前模块, 包含如下属性。

- `id`: 模块的识别符, 通常是带有绝对路径的模块文件名。
- `filename`: 模块的文件名, 带有绝对路径。
- `loaded`: 返回一个布尔值, 表示模块是否已经完成加载。
- `parent`: 返回一个对象, 表示调用该模块的模块。
- `children`: 返回一个数组, 表示该模块要用到的其他模块。
- `exports`: 表示模块对外输出的值。

其中 `exports` 属性是编写模块的关键, 其表示当前模块对外输出的接口。其他文件加载该模块, 实际读取的是 `module.exports`。编写 CommonJS 规范的模块, 实例代码如下:

```

// moduleA.js
module.exports = function(params){
  console.log(params);
}

```

使用方法如下:

```

// 假设两个文件在同一目录下
var moduleA = require('./moduleA');
moduleA();

```

为了方便, Node.js 为每个模块提供一个 `exports` 变量指向 `module.exports`, 那么 `moduleA` 的写法也可以这样编写, 代码如下:

```
// moduleA.js
exports.moduleA = function(params){
    console.log(params);
}
```

注意: 不能把值直接赋给 exports, 因为这样等于切断了 exports 与 module.exports 的联系。

总结 CommonJS 模块的特点如下:

- 所有模块都有单独作用域, 不会污染全局作用域。
- 重复加载模块只会加载第一次, 后面都从缓存读取。
- 模块加载的顺序按照代码中出现的顺序。
- 模块加载是同步的。

2. AMD 规范

CommonJS 模块采用同步加载, 适合服务端却不适合浏览器。AMD 规范支持异步加载模块, 规范中定义了一个全局变量 define 函数, 描述如下所示:

```
define(id?, dependencies?, factory);
```

第一个参数 id, 为字符串类型, 表示模块标识, 为可选参数。若不存在则模块标识默认定义为在加载器中被请求脚本的标识。如果存在, 那么模块标识必须为顶层的或者一个绝对的标识。

第二个参数 dependencies, 定义当前所依赖模块的数组。依赖模块必须根据模块的工厂方法优先级执行, 并且执行的结果按照依赖数组中的位置顺序以参数的形式传入(定义中模块的)工厂方法中。

第三个参数 factory, 为模块初始化要执行的函数或对象。如果为函数, 只被执行一次。如果是对象, 此对象应该为模块的输出值。如果工厂方法返回一个值(对象、函数或任意强制类型转换为 true 的值), 应该设置为该模块的输出值。

- 创建一个标准 AMD 模块

创建模块标识为“alpha”的模块, 依赖于内置的“require”和“exports”模块和外部标识为“beta”的模块。require 函数取得模块的引用, 从而即使模块没有作为参数定义, 也能够被使用。exports 是定义的 alpha 模块的实体, 在其上定义的任何属性和方法也就是 alpha 模块的属性和方法。例子中简单调用模块 beta 的 verb 方法, 代码如下:

```
define("alpha", ["require", "exports", "beta"], function(require, exports, beta) {
    export.verb = function() {
        return beta.verb();
    };
});
```

```
// 或者:
return require("beta").verb();
}
});
```

- 创建一个匿名模块

define 方法允许省略第一个参数, 当省略第一个参数定义模块时, 模块文件的文件名即模块标识, 该模块为匿名模块。定义一个依赖于 **alpha** 模块的匿名模块, 代码如下:

```
define(["alpha"],function(alpha){
    return {
        verb : function(){ return alpha.verb() + 1 ; }
    }
});
```

- 仅有一个参数的 **define**

define 的前两个参数都是可以省略的。第三个参数有两种情况, 一种是 JavaScript 对象, 另一种是函数。

如果参数是一个对象, 那么可能是一个包含方法的对象, 也可能仅提供数据, 或者都存在, 代码如下:

```
define({
    name : 'add',
    add : function( x, y ){ return x + y ; }
});
```

如果参数是一个函数, 其用途是快速开发实现, 适用于较小型的应用, 代码如下:

```
define(function(){
    // 使用 math-util 这个模块
    var mathUtil = require(math-util);
    return mathUtil.add(1,2);
});
```

- 局部 **require** 与全局 **require**

局部 **require** 可以被解析成一个符合 AMD 工厂函数规范的 **require** 函数, 实例代码如下:

```
define(['require'],function(require){
    // ...
});
// 或者:
define(function(require,exports,module){
```

```
// ...
});
```

局部 `require` 也支持其他标准实现的 API。全局 `require` 函数作用于全局，和 `define` 函数类似。全局 `require` 和局部 `require` 有着相同的行为，均包含这些特征：模块 ID 应该认为是一个绝对的模块名称，而不是相对另一个模块的 ID；只有在异步的时候，才可以使用 `require(id, callback)` 的回调形式。因为异步加载模块的方式是先发出一个异步请求，然后等主线程代码段执行完毕才能进行异步回调并处理加载完毕的模块。

实际中，经常会遇到一些阻塞模块加载的依赖，如果交互次数很多，需要大量的模块加载，应该采用全局依赖的形式去加载顶层模块。

• RequireJS 介绍

说到 AMD 规范就不得不说 RequireJS，RequireJS 库能够把 AMD 规范应用到实际浏览器 Web 端的开发中。其主要解决了两个问题：实现 JavaScript 文件的异步加载，避免网页失去响应；管理模块之间的依赖性，便于代码的编写和维护。

首先，官网下载最新版 `require.js` 文件，当前版本为 2.3.3，地址为 <http://requirejs.org/>，并在页面底部引入，实例代码如下：

```
<script src="js/require.js"></script>
```

加载实例逻辑的主模块文件，代码如下：

```
<script src="js/require.js" data-main="js/main"></script>
```

`data-main` 属性定义 Web 程序的主模块，在这里“`js/main`”即主模块，省略了后缀“`.js`”，RequireJS 在加载脚本引用时会为其默认添加。主模块也称为入口文件，类似于 C 语言的 `main` 函数，所有代码都从这儿开始运行。`main.js` 的实例代码如下：

```
require(['jquery', 'underscore', 'backbone'], function($, _, Backbone) {
    // 业务代码
});
```

RequireJS 会依次加载类库 `jQuery`、`Underscore` 和 `Backbone.js`，然后再运行回调函数。使用 `require.config()` 方法，开发者可以对模块的加载路径进行自定义，假设这些库文件都在和 `main.js` 同级的 `lib` 文件夹下，实例代码如下：

```
require.config({
    paths: {
        "jquery": "lib/jquery.min",
        "underscore": "lib/underscore.min",
```



```

    "backbone": "lib/backbone.min"
  }
});

```

或者使用属性 `baseUrl` 定义基础路径, 代码如下:

```

require.config({
  baseUrl: "js/lib",
  paths: {
    "jquery": "jquery.min",
    "underscore": "underscore.min",
    "backbone": "backbone.min"
  }
});

```

RequireJS 支持加载非 AMD 规范的模块, 支持使用 `require.config` 方法来定义一些特征, 比如 Underscore 和 Backbone.js 这两个库 (非 AMD 规范), 实例代码如下:

```

require.config({
  shim: {
    'underscore': {
      exports: '_'
    },
    'backbone': {
      deps: ['underscore', 'jquery'],
      exports: 'Backbone'
    }
  }
});

```

代码中的 `shim` 属性, 专门用来配置不兼容的模块。具体来说, 每个模块要定义: `exports` 值 (输出的变量名), 表明这个模块外部调用时的名称; `deps` 数组, 表明该模块的依赖。

3. CMD 规范

CMD 规范全称为 Common Module Definition, 下面介绍该规范实现的关键函数。

• define 函数

在 CMD 规范中, 一个模块就是一个文件, 书写格式如下:

```
define(factory);
```

`define` 是一个全局函数, 用来定义模块, 接收 `factory` 参数, `factory` 可以是一个函数, 也可以是一个对象或字符串。当 `factory` 参数为对象、字符串时, 表示模块的接口就是该对象、字符串。比如, 定义一个 JSON 数据作为 `factory` 参数, 代码如下:

```
define({ 'foo': 'bar' });
```

也可以通过字符串定义模板模块，代码如下：

```
define('I am a template. My name is {{name}}.');
```

factory 为函数时，表示是模块的构造方法。执行该构造方法，可以得到模块向外提供的接口。**factory** 方法在执行时，默认会传入三个参数：**require**、**exports** 和 **module**，代码如下：

```
define(function(require, exports, module) {
    // 模块代码
});
```

define 也可以接收两个以上的参数，语法如下所示：

```
define(id?, deps?, factory)
```

字符串 **id** 表示模块标识，数组 **deps** 是模块依赖，实例代码如下：

```
define('hello', ['jquery'], function(require, exports, module) {
    // 模块代码
});
```

define.cmd 方法可用来判定当前页面是否有 **CMD** 模块加载器，实例代码如下：

```
if (typeof define === "function" && define.cmd) {
    // 有 Sea.js 等 CMD 模块加载器存在
}
```

• require 函数

require 是一个方法，接收模块标识作为唯一参数，用来获取其他模块提供的接口。实例代码如下：

```
define(function(require, exports) {
    var a = require('./a');           // 获取模块 a 的接口
    a.doSomething();                  // 调用模块 a 的方法
});
```

• exports 对象

exports 是一个对象，用来向外提供模块接口，实例代码如下：

```
define(function(require, exports) {
    exports.foo = 'bar';              // 对外提供 foo 属性
    exports.doSomething = function() {}; // 对外提供 doSomething 方法
});
```

除了给 `exports` 对象增加成员外, 还可以使用 `return` 直接向外提供接口, 实例代码如下:

```
define(function(require) {  
  return {                                // 通过 return 直接提供接口  
    foo: 'bar',  
    doSomething: function() {}  
  };  
});
```

如果 `return` 语句是模块中的唯一代码, 还可简化为如下代码:

```
define({  
  foo: 'bar',  
  doSomething: function() {}  
});
```

Sea.js 作为 CMD 规范的经典实现, 追求简单、自然的代码书写和组织方式, 具有以下核心特性。

- 简单友好的模块定义规范, Sea.js 遵循 CMD 规范, 可以像 Node.js 一样书写模块代码。
- 自然直观的代码组织方式, 依赖自动加载, 配置简洁清晰, 可以让开发者更多地享受编码的乐趣。

Sea.js 还提供常用插件, 帮助开发调试和性能优化, 并具有丰富的可扩展接口。前往 [GitHub](https://github.com/seajs/seajs) 下载最新 `sea.js` 文件, 地址为 <https://github.com/seajs/seajs>, 在页面底部引入该脚本, 配置如下代码:

```
seajs.config({  
  // Sea.js 在解析顶级标识时, 会相对 base 路径来解析  
  base: "../sea-modules/",  
  // 当模块标识很长时, 使用 alias 简化  
  alias: {"jquery": "jquery/jquery/1.10.1/jquery.js"}  
});  
// 加载入口模块  
seajs.use("../src/main")
```

`main.js` 是程序的入口文件, 实例代码如下:

```
// 所有模块都通过 define 来定义  
define(function(require, exports, module) {  
  // 通过 require 引入依赖  
  var $ = require('jquery');  
  // 通过 exports 对外提供接口  
  exports.doSomething = ...  
});
```

```
// 或者通过 module.exports 提供整个接口
module.exports = ...
```

```
});
```

5.6.3 ECMAScript 6 标准的模块支持

ECMAScript 5 及之前的版本不支持原生模块化，需要引入 AMD 规范的 RequireJS 或者 CMD 规范的 Seajs 等第三方库实现。直到 ECMAScript 6 才支持原生模块化，其不但具有 CommonJS 规范和 AMD 规范的优点，而且实现得更加友好，语法较之 CommonJS 更简洁、支持编译时加载或者叫静态加载、循环依赖处理得更加优秀。

ECMAScript 6 模块功能主要由两个命令构成：`export` 和 `import`。`export` 命令用于规定模块的对外接口，`import` 命令用于输入其他模块提供的功能。

1. export

在 ECMAScript 6 中，一个模块也是一个独立的文件，具有独立的作用域，通过 `export` 命令输出内部变量，实例代码如下：

```
// car.js 代码
var name = 'bus';
var color = 'green';
var weight = '20 吨'
export {name, color, weight};
```

`export` 命令除了输出变量，还可以输出函数或类（class），实例代码如下：

```
export function run() { console.log('Bus is running'); };
```

可以使用 `as` 关键字对输出的变量、函数、类重命名，实例代码如下：

```
var name = 'bus';
var color = 'green';
var weight = '20 吨'
function run() { console.log('Bus is running'); }
export {
  name as busName,
  color as busColor,
  weight as busWeight,
  run as busRun
}
```


注意: `export` 命令规定的是对外的接口, 必须与模块内部的变量建立一一对应关系。

以下导出方式会报错, 代码如下:

```
// 报错
export 20;
// 报错
var name = 'bus';
export name;
```

正确的使用方式代码如下:

```
var name = 'bus';
export {name}
```

因为无论是“20”还是“bus”都是具体的值, 不是接口。同样的, `function` 和 `class` 的输出, 也必须遵守这样的写法。

2. import

`import` 命令用于导入模块, 实例代码如下:

```
import {name, color, weight, run} from './car';
```

导入模块的时候也可以使用 `as` 关键字对模块进行重命名, 实例代码如下:

```
import {name as busName} from './car';
```

如果一个模块没有输出, 也可以用于只加载其他模块, 实例代码如下:

```
import lodash;
```

上述代码仅仅是加载了 `lodash` 模块, 没有任何输入的值。重复加载模块也只会执行一次。

可以通过星号“*”整体加载某个文件, 实例代码如下:

```
import * as car from './car';
console.log(car.name);           // 输出结果: bus
console.log(car.color);          // 输出结果: green
```

3. export default 命令

从前面的例子可以看出, 使用 `import` 命令加载模块时需要知道变量名或者函数名, 或者整个文件, 否则无法加载。为了方便, 可以使用 `export default` 命名为模块指定默认输出。实例代码如下:

```
export default function() { console.log('default');};
```

加载该模块时, 可使用 `import` 命名为其指定任意名字, 实例代码如下:

```
import someName from './export-default';
someName ();           // 即执行"default"方法
```

以上就是 ECMAScript 6 模块化的基本使用方法了，读者还可以通过 <https://github.com/tc39/ecma262> 学习更详细的内容。

注意：现在并非所有的浏览器都支持 ECMAScript 6 语法，如果要在现代浏览器中使用 ECMAScript 6，需要借助 Babel 编译器，具体使用方法可以参考 Babel 的官网 <http://babeljs.io/>。

5.7 ECMAScript 6 其他常用功能

前面章节介绍了 ECMAScript 5 语言的基础功能，以及部分 ECMAScript 6 语言的重要特性。本节将介绍 ECMAScript 6 语言新增的基础数据类型，解构赋值的应用，以及如何使用 Babel 来支持 ECMAScript 6 语言新特性。

5.7.1 基础数据类型的扩展

ECMAScript 6 不但对老的基础数据类型进行了扩展，而且增加了一个全新的基础数据类型 Symbol。接下来介绍常用的一些扩展功能。

1. 字符串的扩展

ECMAScript 6 新增的常用字符串函数包括 `startsWith`、`endsWith`、`includes` 和 `repeat`，实例代码如下：

```
var s = "Hello";
s.startsWith("He"); // 返回 true，表示源字符串是否以参数字符串开始
s.endsWith("lo");   // 返回 true，表示源字符串是否以参数字符串结束
s.includes("el");    // 返回 true，表示源字符串是否包含参数字符串
s.repeat(3);         // 返回"HelloHelloHello"，表示将源字符串重复 3 次
```

ECMAScript 6 引入模板字符串来简化字符串拼接。传统字符串拼接的代码如下：

```
var firstName = "Jet";
var lastName = "Li";
var sentence = "My first name is " + firstName + ", and my family name is " + lastName;
```

使用模板字符串的代码如下：

```
var firstName = "Jet";
```

```
var lastName = "Li";
var sentence = `My first name is ${firstName}, and my family name is ${lastName}`;
```

模板字符串不但可以使用变量, 还可以使用表达式和调用函数, 代码如下:

```
function minus (x, y) { return x - y; } // 实现减法功能
var x = 5;
var y = 3;
var result1 = `5 minus 3 equals ${x - y}`; // 返回 2
var result2 = `5 minus 3 equals ${minus(x, y)}`; // 返回 2
```

2. 数值的扩展

ECMAScript 6 规范了二进制和八进制的表示方法, 代码如下:

```
0o2000 === 1024 // 使用 0o 表示八进制
0b1000000000000 === 1024 // 使用 0b 表示二进制
```

ECMAScript 6 将全局函数 `parseInt` 和 `parseFloat` 移植到了 `Number` 对象上, 并且在 `Number` 对象上增加了 `isNaN` 和 `isInteger` 方法, 代码如下:

```
// Number.isNaN 判断参数的值是否是 NaN
Number.isNaN(NaN); // 返回 true
Number.isNaN(1024); // 返回 false
Number.isNaN("1024"); // 返回 false
Number.isNaN(true); // 返回 false
// Number.isInteger 判断参数的值是否是整数
Number.isInteger(25) // 返回 true
Number.isInteger(25.0) // 返回 true
Number.isInteger(25.1) // 返回 false
Number.isInteger("25") // 返回 false
Number.isInteger(true) // 返回 false
```

注意: 在 JavaScript 内部, 整数和浮点数是同样的存储方法, 所以 25.0 会被存储为 25。

`Number` 对象增加了一个极小常量 `EPSILON`。由于浮点数的计算是不精确的, 会导致无法判断浮点数的运算结果是否等于某个值, 代码如下:

```
(0.1 + 0.2) === 0.3 // 返回 false
```

这个时候, 如果两个值的误差小于极小常量 `EPSILON`, 就认为两个值相等, 代码如下:

```
Math.abs(0.1 + 0.2 - 0.3) < Number.EPSILON // 返回 true
```

3. 函数的扩展

ECMAScript 6 可以给函数参数指定默认值，代码如下：

```
function minus (x, y = 2) { return x - y; }
minus(5); // 返回结果: 3
```

ECMAScript 6 引入了 Rest 语法来获取函数剩余参数，即处理可变参数的函数，代码如下：

```
function add (...values) {
  let r = 0;
  for (let n of values) { r += n; }
  return r;
}
add(1, 2, 3, 4); // 返回 10
```

ECMAScript 6 使用箭头函数来简化函数的定义，代码如下：

```
// ECMAScript 5 函数的定义
[1, 2, 3].map(function(x) { return x + 1; });
// 箭头函数的定义
[1, 2, 3].map(x => x + 1);
```

注意：箭头函数内部 this 对象指向定义时的外部环境中的 this 对象。

4. 对象的扩展

ECMAScript 6 支持对象定义时属性和函数的简写，并且属性名支持表达式，代码如下：

```
// ECMAScript 5 属性和函数定义
var p = 1;
var o = {
  p: p,
  f: function () { }
}
// ECMAScript 6 简化属性和函数定义，并且属性名支持表达式
var p = 1;
var o = {
  p,
  ["a" + p]: 2, // 属性名为 a1
  f () { }
}
```

5. 新增 Symbol 基础类型

Symbol 代表独一无二的值，可以用来避免冲突。Symbol 类型出现之前申明枚举一般通过使用

字符串作为值,代码如下:

```
var color = {
  red: 'red',
  green: 'green'
}

var selectedColor = color.red;
selectedColor === color.red;           // 返回 true
selectedColor = "red";
selectedColor === color.red;           // 返回 true
```

上述代码潜在的风险是当要比较的值正好也是一个字符串时,有可能会误判为等于枚举类型的值,使用 `Symbol` 类型就不会有这种风险,代码如下:

```
var color = {
  red: Symbol(),
  green: Symbol()
}

var selectedColor = color.red;
selectedColor === color.red;           // 返回 true
selectedColor = "red";
selectedColor === color.red;           // 返回 false
```

5.7.2 使用解构赋值来简化代码

ECMAScript 6 允许按照一定模式,从数据结构中提取值,对变量进行赋值,这个过程被称为解构。ECMAScript 6 对常用的数据结构提供了解构赋值的支持,以达到简化代码的目的。支持解构赋值的类型包括对象、数组、字符串、函数参数等。

1. 对象的解构赋值

普通解构赋值,实例代码如下:

```
var o = { firstName: 'Jet', lastName: 'Li' };
var { firstName, lastName } = o;           // firstName 值为"Jet", lastName 值为"Li"
var { lastName, firstName } = o;           // firstName 值为"Jet", lastName 值为"Li"
var { middleName } = o;                     // middleName 值为 undefined
```

允许属性名自定义,代码如下:

```
var o = { firstName: 'Jet', lastName: 'Li' };
var { lastName: familyName } = o;           // familyName 值为"Li"
```

属性可以有默认值，代码如下：

```
var o = { firstName: 'Jet', lastName: 'Li' };
// middleName 值为"super", lastName 值为"Li"
var { middleName = "super", lastName="start" }
```

对象解构赋值可以嵌套，代码如下：

```
var o = {sex: 'male', name: {firstName: 'Jet', lastName: 'Li'}};
// sex 值为"male", firstName 值为"Jet", lastName 值为"Li", name 值为 undefined。
var { sex, name: {firstName, lastName} } = o;
```

注意：这里的 name 不会被赋值。

2. 数组的解构赋值

普通解构赋值，代码如下：

```
var arr = ["Jet", "Li"];
// firstName 值为"Jet", lastName 值为"Li", middleName 值为 undefined。
var [firstName, lastName, middleName] = arr;
```

属性可以有默认值，代码如下：

```
var arr = ["Jet", "Li"];
// firstName 值为"Jet", lastName 值为"Li", middleName 值为 super。
var [firstName, lastName, middleName = "super"] = arr;
```

数组解构赋值可以嵌套，代码如下：

```
var arr = ["male", ["Jet", "Li"]];
// sex 的值为"male", firstName 值为"Jet", lastName 值为"Li"。
var [sex, [firstName, lastName]] = arr;
```

可以看到，数组是按照顺序解构，对象是按照属性名解构。除此之外，两者的使用方式是一致的。

数组本身也是对象，也可以按照对象的方式解构，代码如下：

```
var arr = ["Jet", "Li"];
// firstName 值为"Jet", lastName 值为"Li", len 值为 2。
var {length: len, "0": firstName, "1": lastName} = arr;
```

3. 字符串的解构赋值

字符串使用数组方式解构，代码如下：

```
// a, b, c, d, e 的值分别为"h","e","l","l","o"
```

```
var [a, b, c, d, e] = "Hello";
```

字符串使用对象方式解构，代码如下：

```
var {length: len, "0": firstLetter} // len 的值为"5", firstLetter 的值为"H"
```

4. 其他数据结构的解构赋值

任何数据结构都可以使用对象方式解构，代码如下：

```
var {toString: s} = 1024;
s === Number.prototype.toString // 返回结果: true
```

任何实现了 Iterator 接口的数据结构都可以使用数组方式解构，代码如下：

```
// firstName 值为"Jet", lastName 值为"Li"
var [firstName, lastName] = new Set(["Jet", "Li"]);
```

Set 是 ECMAScript 6 新增的数据结构，可以实现 Iterator 接口。

5. 函数参数的解构赋值

除了变量赋值的时候支持解构，函数参数赋值同样支持解构，两者对解构的支持程度完全一样，代码如下：

```
function minus ([x, y]) { return x - y; }
minus([5, 3]); // 返回 2
function minus ({x, y}) { return x - y; }
minus({x: 5, y: 3}); // 返回 2
```

5.7.3 使用 Babel 插件提前使用新特性

ECMAScript 6 在每个浏览器的支持程度不同，为了能够在项目开发过程中提前使用，需要将书写的 ECMAScript 6 代码转换为现代浏览器可以直接运行的 ECMAScript 5 代码，而 Babel 就是一个被广泛使用的转码器。

1. 使用 Babel

在使用 Babel 前，首先将 Babel 安装到项目中。

(1) Babel 支持 NPM 包形式的安装，打开命令行窗口，切换到项目根目录，命令如下：

```
npm install babel-cli
```

(2) 安装成功后，在 package.json 文件里添加如下代码：

```
{
```

```

"scripts": {
  "start": "babel test.js --out-file test-compiled.js"
}

```

(3) 创建一个使用了 ECMAScript 6 语法的 JavaScript 文件 test.js, 输入如下代码:

```
[1, 2, 3].map(n => n + 1);
```

(4) test.js 文件里的代码使用了 ECMAScript 6 的箭头函数, 需要使用 Babel 转码。打开命令行窗口, 输入如下命令:

```
npm run start
```

(5) 编译过后的代码输出到了 test-compiled.js, 代码如下:

```
[1, 2, 3].map(n => n + 1);
```

(6) 编译后的代码没有变化, Babel 没有将箭头函数转换成普通函数。Babel 默认不对任何语法转换, 需要事先通过配置文件来指定转码 ECMAScript 6 语法特性。创建配置文件 “.babelrc”, 代码如下:

```

{
  "plugins": [
    "transform-es2015-arrow-functions"
  ]
}

```

(7) 安装 Babel 转换插件。打开命令行窗口, 输入如下命令:

```
npm install babel-plugin-transform-es2015-arrow-functions
```

注意: 配置文件中插件的名字跟安装的 NPM 包的名字并不相同, 配置文件内会省略后缀 “babel-plugin”。

(8) 再次使用 Babel 进行转码。打开命令行窗口, 输入如下命令:

```
npm run start
```

打开转码后的文件 test-compiled.js, 里面包含如下代码:

```

"use strict";
[1, 2, 3].map(function (n) { return n + 1; });

```

箭头函数已经被转换成了普通函数, 转换后的代码可以直接在浏览器中运行。

插件 “transform-es2015-arrow-functions” 只负责对箭头函数转码, 如果需要转码其他的

ECMAScript 6 特性, 需添加对应的插件到配置文件。Babel 插件列表可以通过访问页面 <http://babeljs.io/docs/plugins> 查看, 里面包含了现有插件以及每个插件的使用方式和转码效果。

2. 使用 Babel 插件集

通常在开发时, 大部分的 ECMAScript 6 的语法都会用到, 手动配置所有需要的插件是一件非常麻烦而且容易出错的工作。Babel 在插件机制之上, 提供了插件集, 每个插件集包含相关的一系列插件。

(1) 打开配置文件 “.babelrc”, 配置插件集, 代码如下:

```
{  
  "presets": [ "latest" ]  
}
```

(2) 安装 Babel 插件集。打开命令行窗口, 输入如下命令:

```
npm install babel-preset-latest
```

(3) 开发者需要注意的是, Babel 只对 ECMAScript 6 的新语法进行转码, 不会转换新的 API, 比如不支持转码 Object.assign 函数。如果开发中用到新的 API, 可以使用 babel-polyfill 添加垫片。安装垫片代码如下:

```
npm install babel-polyfill
```

(4) 安装成功之后, 在入口脚本文件头部加入垫片。代码如下:

```
import "babel-polyfill";
```

(5) 如果在 Node.js 环境里运行 JavaScript 文件, 除了可以使用命令在运行前进行转码外, 也可以使用钩子在运行时进行转码。创建 server.js 文件, 输入如下代码:

```
import "http";  
console.log("Hello World");
```

(6) 打开命令行窗口, 输入命令, 运行 server.js, 命令如下:

```
node server.js
```

因为 Node.js 不支持 ECMAScript 6 的 import 语法, 会出现报错信息 “Unexpected token import”。

(7) 打开命令行窗口, 安装钩子, 命令如下:

```
npm install babel-register
```

(8) 创建 hook.js 文件, 注册钩子, 代码如下:

```
require("babel-register");
require("../server");
```

(9) 运行钩子文件，打开命令行窗口，输入如下命令：

```
node hook.js
```

成功打印信息“Hello World”。

提示：Babel 官网提供了一个在线转换页面 <https://babeljs.io/repl/>，可以实时查看 ECMAScript 6 转码成的 ECMAScript 5 格式，还可以配置不同的插件及插件集，对学习 ECMAScript 6 以及 Babel 非常有帮助。

5.8 本章小结

JavaScript 语言作为一门诞生于浏览器的脚本语言，在 Web 页面中承担着越来越大的作用，是实现复杂页面交互的技术基础。而且随着 Node.js、React Native 变成热门，JavaScript 语言已经开始向服务端和 APP Native 端迁移，未来 JavaScript 语言会出现更广阔的使用场景。

本章的开始，介绍了 JavaScript 的语言基础及事件，随后深入地了解了作用域及闭包概念，学习如何使用 JavaScript 进行面向对象编程及对异步的高级处理，最后学习了在实际项目中非常重要的模块化及其他一些常用的功能。本章的内容，需要熟练掌握，是以后学习更高阶开发的基础。

6

第 6 章

HTML 5 移动开发实战

HTML 5 从第一份草案开始到最终的正式版，历经了长达 6 年多时间。在这 6 年多的时间里，互联网在国内发生了翻天覆地的变化，不论是在技术还是商业上都到达了空前的高度，用户群体也开始从最初的 PC 端渐渐地转移至移动端。非开发人员口中也常常出现“H5”相关的词汇，俗称的“H5”即 HTML 5，足以见得这门技术在移动开发中的重要程度。

本章将通过多个实际开发场景，如地理定位、在线聊天、拍摄、播放器、动画、3D 等，介绍 HTML 5 实战开发。

6.1 在地图上显示行走轨迹

本节将实现在地图上记录用户历史运动轨迹的功能，如图 6.1 所示。

打开页面，载入地图，单击“开始记录”按钮，随着用户的移动，同步在地图上呈现历史行动轨迹，单击“停止记录”按钮，停止记录轨迹，并清除历史记录轨迹。



图 6.1 高德地图显示运动轨迹

代码实现步骤如下。

(1) 本实例采用高德地图，需要在高德地图官网上申请 AppKey，然后引入高德地图的 JavaScript，代码如下：

```
<script src="//webapi.amap.com/maps?v=1.3&key=申请的高德地图 Key"></script>
```

(2) 在页面中插入 HTML 结构，代码如下：

```
<header>                                <!-- 控制记录轨迹的按钮 -->
  <button id="btnStart">开始记录</button>
  <button id="btnStop">停止记录</button>
</header>
<div id="map"></div>                    <!-- 地图容器 -->
<script type="text/javascript" src="geo.js"></script>    <!-- 页面渲染地图脚本-->
```

(3) 在上述代码的 geo.js 文件中，实现了整个实例的所有代码。首先，调用高德地图 API 绘制地图，并设置地图的中心点和较低的缩放级别，显示整个城市的地图，代码如下：

```
var map = new AMap.Map('map', {
  center: [121.600000, 31.220000],          // 地图中心点
  zoom: 10                                  // 默认的放大级别
});
map.plugin(["AMap.ToolBar"], function () {  // 给地图增加工具条，控制地图的放大和缩小
  map.addControl(new AMap.ToolBar());
});
```

通过 AMap.Map 构造函数构建地图对象，代码如下：

```
AMap.Map(container, options)
```

- container: 地图容器元素的 ID 或者 DOM 对象；

- options: 地图配置项, 具体参考高德地图 API。

注意: 在高德地图中的经纬坐标系采用了“火星坐标”(GCJ-02), 也称为国测局坐标系, 与 GPS 所采用的“地球坐标”(WGS84)不同。通过 GPS 定位得到的经纬度需要换算才能用于高德地图。高德官方提供了 API, 以实现坐标换算。

(4) 接着介绍通过 HTML 5 的地理信息接口获取当前的地理位置, 代码如下:

```
01 var geoOptions = {
02     enableHighAccuracy: true,          // 开启高精度 (GPS) 定位
03     timeout: 30000,                    // 设置接口超时时间为 30s
04     maximumAge: 1000                   // 设置地理信息的最大缓存时间为 1s
05 }
06 function getPosition(callback) {
07     if (navigator.geolocation) {
08         navigator.geolocation.getCurrentPosition(function (position) {
09             callback(position.coords);
10         }, function (error) {
11             switch (error.code) {
12                 case 0:
13                     alert("尝试获取您的位置信息时发生错误: " + error.message);
14                     break;
15                 case 1:
16                     alert("用户拒绝了获取位置信息请求。");
17                     break;
18                 case 2:
19                     alert("浏览器无法获取您的位置信息。");
20                     break;
21                 case 3:
22                     alert("获取您位置信息超时。");
23                     break;
24             }
25         }, geoOptions);
26     }
27 }
```

上述代码中, 定义了 `getPosition` 函数, 函数中调用 `navigator.geolocation.getCurrentPosition` 接口, 获取当前地理位置, 该接口的定义如下:

```
getCurrentPosition(successCallback, errorCallback, options)
```

- successCallback: 获取地理信息成功的回调函数, 该参数必选。

- **errorCallback**: 获取地理信息失败的回调函数, 该参数可选。
- **options**: 调用接口的配置项, 该参数可选。其中包含属性 **enableHighAccuracy** 表示是否启用高精度定位 (开启 GPS), 默认值为 **false**; 属性 **timeout** 表示接口的超时时间, 单位为 **ms** (毫秒), 默认值为 **Infinity**; 属性 **maximumAge** 表示地理信息的最大缓存时间, 单位为 **ms** (毫秒), 默认值为 **0**。

其中 **getCurrentPosition** 接口返回的 **successCallback** 成功回调函数的参数描述如下。

- **coords**: 地理定位信息。
- **coords.latitude**: 十进制的纬度估值, 范围为 $[-90, 90]$ 。
- **coords.longitude**: 十进制的经度估值, 范围为 $[-180, 180]$ 。
- **coords.altitude**: 海拔, 单位为米, 可选。
- **coords.accuracy**: 经纬度的精度, 单位为米, 该值越小, 表示定位越精准, 可选。
- **coords.altitudeAccuracy**: 海拔的精准度, 单位为米, 类似于经纬度的精准度, 可选。
- **coords.heading**: 前进的方向, 度数表示, 相对正北方向设备以顺时针方向运动计算的当前方向, 可选。
- **coords.speed**: 前进的速度, 单位为米/秒, 可选。
- **timestamp**: 成功响应时的时间戳。

(5) 在本实例中, 由于要记录用户的运动轨迹, 因此需要获取高精度位置, 所以将 **options.enableHighAccuracy** 设置为 **true**。

注意: 地理信息接口在不少浏览器中要求页面必须采用 **HTTPS** 协议, 否则无法调用。实测发现, 在非 **HTTPS** 情况下, **Chrome** 可以得到定位的权限, **Safari** 无法得到定位的权限。

(6) 在页面加载完毕后, 调用定义的 **getPosition** 方法, 获取当前地理位置, 代码如下:

```
getPosition(function (coords) {
    coords = convert(coords.longitude, coords.latitude);    // 转换地理坐标
    // 依据坐标, 生成高德地图点对象
    var startPoint = new AMap.LngLat(coords.longitude, coords.latitude);
    map.setCenter(startPoint);    // 设置地图的中心点
    map.setZoom(16);    // 放大地图显示
});
```

(7) 获取地理信息之后, 设置当前位置为地图中心点, 并放大地图。单击“开始记录”按钮, 程序开始记录用户移动轨迹, 代码如下:

```

01 function start() {
02     timmer = navigator.geolocation.watchPosition(function (position) {
03         var coords = position.coords;
04         if (coords.accuracy > 20) { // 去掉低精度的定位数据
05             return;
06         }
07         coords = convert(coords.longitude, coords.latitude); // 转换坐标信息
08         map.setCenter(new AMap.LngLat(coords.longitude, coords.latitude));
09         lineArr.push([coords.longitude, coords.latitude]);
10         renderTracer(getPath(lineArr)); // 调用方法在地图上画轨迹
11     }, function (error) {
12         console.log(error)
13     }, geoOptions); // 该 options 和获取定位的接口一致
14 }

```

采用 `navigator.geolocation.watchPosition` 接口, 监听位置信息的变化, 得到更新的经纬度信息, 去掉低精度数据以避免绘制轨迹时, 轨迹线存在较大误差。该接口的参数和 `getCurrentPosition` 接口一致。在获取定位数据的时候, 可以依据实际情况, 去掉定位精准度较低的数据。

注意: `watchPosition` 方法在非 HTTPS 的场景下, 无法获取定位权限。在 Chrome 下, 可以先通过 `getCurrentPosition` 方法获取定位权限。

限于篇幅, 这里就不细致介绍绘制轨迹的方法, 完整代码请参考本书提供的源码。

本节介绍了通过 HTML 5 的 `navigator.geolocation.getCurrentPosition` 获取当前地理信息, 使用 `navigator.geolocation.watchPosition` 方法监听地理信息的变化来显示行走轨迹。在实际开发中, 建议采用 HTTPS 协议, 以得到更好的体验。

6.2 仿原生相册

本节将实现一个简易版的仿原生相册, 默认界面如图 6.2 所示。单击图片进入全屏观看模式, 向左或向右滑动切换下一张或上一张。手指向左或向右拖曳图片, 图片可随手指运动, 如图 6.3 所示。



图 6.2 相册默认界面

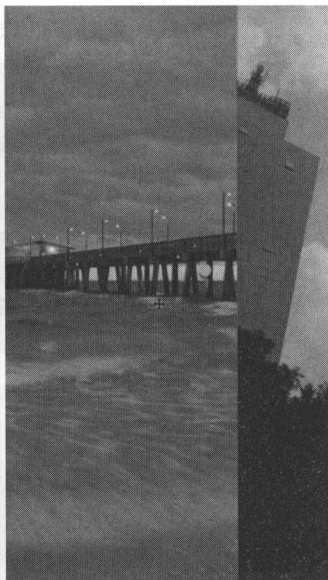


图 6.3 手指拖曳效果

6.2.1 实现相册初始展示页

(1) 设计图 6.2 的初始展示界面，在 Header 标签中添加 Viewport 设置，代码如下：

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

(2) 在 Body 元素中添加小相片列表，代码如下：

```
<div class="gallery">
  <div class="item"></div>
  <div class="item"></div>
  /* 省略其他图片，参见附带源码 */
</div>
```

(3) 添加相册样式，代码如下：

```
.gallery{
  width:100vw;                /* 设置相册的宽度为屏幕宽度 */
  display: flex;              /* 相册采用 flex 布局 */
  flex-flow: row wrap;        /* 相册每一项为横向排列，并且换行 */
}
.gallery .item{ flex: 1; }    /* 每一项平均排列 */
.gallery .item img{ width:33vw; } /* 图片宽度设置为 1/3 屏幕宽度 */
```


(4) 添加预览模式下的效果, CSS 代码如下:

```
.gallery.preview .item{
    display: flex;           /* 对子项设置为 flex 布局 */
    margin: auto;           /* 设置 margin 为 auto 实现图片居中显示 */
}
.gallery.preview .item img {
    max-width: 100vw;       /* 设置预览图片的最大宽度为屏幕宽度 */
    max-height: 100vh;     /* 设置预览图片的最大高度为屏幕高度 */
    width: initial;         /* 初始化图片宽度, 覆盖之前设置的宽度 */
}
```

(5) 在 HTML 页面中加入脚本, 监听相册的 Click 事件, 用户单击相片时, 将相册切换到预览模式, 代码如下:

```
var $gallery = document.querySelector(".gallery");
$gallery.addEventListener("click", function (e) {
    // 监听单击事件, 切换相册的 CSS class 来实现预览和普通模式的切换
    var classList = $gallery.classList,
        css_preview = "preview";
    if (classList.contains(css_preview)) {
        classList.remove(css_preview)
        // 在非预览模式下, 相册的宽度为 100vw
        $gallery.style.width = 100 + "vw"
    } else {
        classList.add(css_preview);
        // 在预览模式下, 所有的图片横着拍成一行, 相册的总宽度为每一个项目长度总和
        $gallery.style.width = 100 * itemsLength + "vw"
    }
});
```

在预览模式下, 通过设置 `gallery` 样式类元素的宽度, 让相册图片排成一行。然后通过 CSS3 的 `Transform` 属性, 设置元素的偏移量, 移动整个元素位置, 使得需要展示的图片出现在屏幕主区域。

6.2.2 通过手势操作控制相片

(1) 在 CSS 中禁用浏览器默认的触摸行为, 代码如下:

```
.gallery.preview{
    touch-action: none;
}
```

(2) 通过监听 `touchstart`、`touchmove` 和 `touchend` 事件来实现滑动手势功能，在 `touchstart` 事件中，记录当前手指的位置，代码如下：

```
$gallery.addEventListener("touchstart", function (e) {
    // 触摸开始时，记住当前手指的位置
    startOffsetX = e.changedTouches[0].pageX;
    // 手指滑动的时候，禁止动画
    $gallery.classList.remove("animation");
});
```

(3) 在 `touchmove` 事件中，使图片随手指拖曳移动，代码如下：

```
$gallery.addEventListener("touchmove", function (e) {
    isTtouchstart = true;
    // 计算手指的水平移动量
    var dx = e.changedTouches[0].pageX - startOffsetX;
    // 调用 move 方法，设置 gallery 元素的 transform，移动图片
    move(currentTranX + dx);
});
```

(4) 图片位置移动采用 CSS3 的 `Transform` 属性实现，代码如下：

```
function (dx) {
    $gallery.style.transform = "translate(" + dx + "px, 0)";
};
```

(5) 在 `touchend` 事件中，判断手指移动的范围是否超过图片宽度的一半。如果达到一半，则切换图片，否则还原图片的位置，代码如下：

```
$gallery.addEventListener("touchend", function (e) {
    if (isTtouchstart) {
        // 在移动图片的时候，需要动画，动画采用 CSS3 的 transition 实现
        $gallery.classList.add("animation");
        var dx = e.changedTouches[0].pageX - startOffsetX;    // 计算偏移量
        // 如果偏移量超出 gallery 宽度的一半
        if (Math.abs(dx) > width / 2) {
            // 处理临界值
            if (currentTranX <= 0 && currentTranX > -width * itemsLength) {
                if (dx > 0) {
                    // 如果手指向右滑动
                    if (currentTranX < 0) {
                        // 如果图片不是显示第一张
                        currentTranX = currentTranX + width;
                    }
                }
                // 如果手指向右滑动，并且当前图片不是显示最后一张
            }
        }
    }
});
```

```

    } else if (currentTranX > -width * (itemsLength - 1)) {
        currentTranX = currentTranX - width;
    }
}
// 如果未超出图片宽度的一半, 拖曳停止, 将图片的位置还原
// 如果超出了图片宽度的一半, 将切换到上一张或下一张图片
move(currentTranX);
}
isTouchstart = false;
});

```

在本实例中, 通过 Flex 布局展示小图片, 通过 touchstart、touchmove 和 touchend 事件, 实现左滑和右滑操作。本例演示了手势操作实现, 但是不同的浏览器调用 touch 事件的方式存在差异, 如在 Internet Explorer 和 Edge 中, 采用了 pointer 事件来处理。社区中存在已经封装的类库, 比如 Pressure.js, GitHub 地址为 <https://github.com/stuyam/pressure>, 大家可以参考。

6.3 使用 Socket.IO 制作小型聊天室

本节将使用 Socket.IO 制作一个小型聊天室。当游客进入聊天室时, 系统会给游客分配一个昵称, 并群发欢迎消息。同样的, 当游客离开聊天室时, 系统群发消息提示。聊天室中的用户可以发送消息, 同时也可收到其他用户信息。

(1) 打开命令行工具, 切换目录至本节源代码所在根目录, 使用 NPM 安装依赖包, 命令如下:

```
npm install
```

(2) 安装成功之后, 启动程序运行如下命令:

```
npm run
```

(3) 成功启动, 控制台打印信息如下:

```
listen at port http://localhost:8080"
```

(4) 使用 Chrome 浏览器访问 <http://localhost:8080/index.html>, 如图 6.4 所示。

注意: 如果要在手机上访问页面, 需要将 localhost 替换为电脑本地 IP 地址, 并确保电脑跟手机在同一局域网。

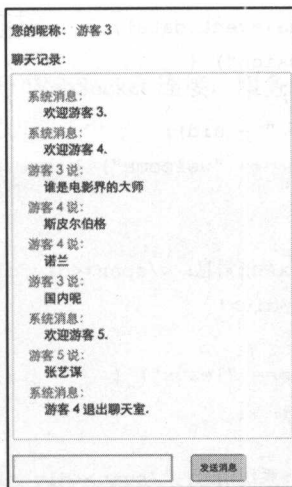


图 6.4 聊天室界面

6.3.1 前端 HTML+JavaScript 实现聊天界面

HTML 核心代码如下:

```
<!-- 省略样式代码, 详见光盘源码 -->
<div class="row">
    您的昵称: <span class="name"></span>
</div>
<div class="row">
    聊天记录: <div class="chat"></div>
</div>
<div class="row">
    <input type="text" class="typed-message"/>
    <input type="button" value="发送消息" class="send-message" />
</div>
```

JavaScript 核心代码如下:

```
01 // 此代码依赖 Zepto.js
02 var ws = new WebSocket("ws://localhost:3000"); // 创建一个 WebSocket 实例
03 var uid; // 最新可分配的游客 ID
04 var $nickname = $(".name"); // 获取昵称元素
05 var $chat = $(".chat"); // 获取聊天记录元素
06 var typedMessage = $(".typed-message"); // 获取输入消息元素
07 ws.onmessage = function(event) { // 当 WebSocket 收到消息时
```



```

08     var message = JSON.parse(event.data);           // 反序列化消息
09     if(message.type === "assign") {                 // 进入聊天室, 分配 ID
10         uid = message.uid;                         // 保存分配的 ID
11         $nickname.text("游客 " + uid);             // 显示昵称
12     } else if (message.type === "welcome") {        // 欢迎消息
13         // 显示消息
14         $chat.append(
15             '<span class="user">系统消息: </span><div class="message">欢迎游客 '
16             + message.uid + ' </div>'
17         );
18     } else if (message.type === "leave") {          // 退出消息
19         // 显示消息
20         $chat.append(
21             '<span class="user">系统消息: </span><div class="message">游客 '
22             + message.uid + ' 退出聊天室.</div>'
23         );
24     } else if (message.type === "message") {        // 用户消息
25         // 显示消息
26         $chat.append(
27             '<span class="user">游客 ' + message.uid + ' 说: </span><div class="message">'
28             + message.message + '</div>'
29         );
30     }
31 }
32 $('send-message').click(function(){               // 发送消息按钮被单击
33     var message = typedMessage.val();              // 获取用户输入的消息
34     if(message === ""){                            // 如果没有输入消息, 直接返回
35         return ;
36     }
37     ws.send(JSON.stringify({                       // 发送消息
38         uid: uid,                                  // 游客 ID
39         message: message                           // 消息
40     }));
41     typedMessage.val('');                          // 清空消息输入框
42 });

```

代码第 07 行通过 WebSocket 对象实例监听服务端消息事件。

代码第 37~40 行将游客输入的消息与游客 ID 发送回服务器。

6.3.2 服务器端 Node.js 监听连接

当退出页面时，浏览器会自动断开 WebSocket 连接，服务端可以监听断开、连接事件。服务器端 Node.js 核心代码如下：

```

01 var WebSocketServer = require('ws').Server;           // 用于 Websocket 服务器
02 var uid = 1;                                           // 当前可分配的用户 id
03 var wss = new WebSocketServer({port: 3000});          // 监听 3000 端口
04 wss.on('connection', function(ws) {                  // 客户端发起连接
05     ws.uid = uid;                                       // 保存分配给游客的 ID
06     wss.broadcast({                                    // 新用户信息告诉房间内所有用户
07         uid: uid,                                       // 游客 ID
08         type: "welcome"                                // 欢迎游客进入
09     });
10     ws.send(JSON.stringify({                          // 发送消息给连接的游客
11         uid: uid,                                       // 分配给游客的 ID
12         type: "assign"                                  // 分配 ID 给游客
13     }));
14     uid++;                                              // 设置下一个可分配 ID
15     ws.on('close', function() {                       // 客户端关闭连接
16         wss.broadcast({                                // 发送消息
17             uid: ws.uid,                                // 离开聊天室的游客 ID
18             type: "leave"                               // 游客离开聊天室
19         });
20     });
21     ws.on('message', function(data) {                  // 客户端发送消息
22         var data = JSON.parse(data);                   // 反序列化消息
23         wss.broadcast({                                // 群发消息给客户端
24             uid: data.uid,                              // 发送消息的游客 ID
25             message: data.message,                      // 游客发送的消息
26             type: "message"                             // 游客发送消息
27         });
28     });
29 });
30 wss.broadcast = function(data) {                      // 群发消息给客户端
31     wss.clients.forEach(function each(client) {        // 遍历客户端列表
32         client.send(JSON.stringify(data));              // 发送消息
33     });
34 }

```

代码第 03 行启动 WebSocket 服务，代码第 04 行监听客户端发起的连接服务事件。

代码第 05~09 行群发消息欢迎新游客的加入，代码第 10~13 行将分配给游客的 ID 发送给新加入的游客。

代码第 15~20 行监听客户端关闭连接的事件，并群发游客离开的消息。

代码第 21~28 行监听游客发送的消息，并将接收的消息进行群发。

6.4 移动端拍照上传实践

本节将实现移动端用户拍照，并将照片上传至服务器。功能包括拍照上传、实时进度显示和图片预览。

(1) 打开命令行工具，切换目录至本节源代码所在根目录，使用 NPM 安装依赖包，命令如下：

```
npm install
```

(2) 安装成功之后，运行如下命令启动程序：

```
npm run
```

(3) 启动成功之后，控制台打印出如下信息：

```
listen at port http://localhost:8080"
```

(4) 在浏览器里输入 `http://localhost:8080/index.html` 即可访问页面，效果如图 6.5 所示。

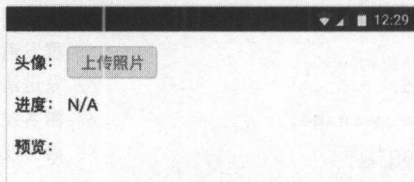


图 6.5 初始上传界面

注意：如果要在手机上访问页面，则将 `localhost` 替换为电脑本地 IP 地址，并确保电脑跟手机在同一局域网。

(5) 单击“上传照片”按钮，浏览器弹窗询问照片来源，如图 6.6 所示。

(6) 单击“拍照或录像”按钮，使用系统摄像头拍照。拍照完毕之后，照片开始自动上传，同时显示预览照片。在上传过程中，可以看到以百分比形式显示的实时进度。待上传完毕之后，

进度栏会显示为“上传成功”或者“上传失败”，如图 6.7 所示。

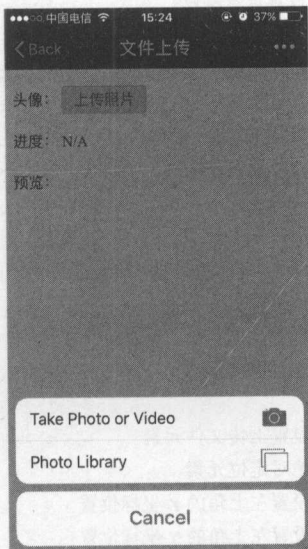


图 6.6 单击“上传照片”按钮系统询问照片来源



图 6.7 上传成功后的界面

6.4.1 前端 HTML+CSS+JavaScript

HTML 核心代码如下：

```
<div class="row">
  头像:
  <a href="javascript:void(0)" class="file">
    <input id="photo" type="file"/>上传照片
  </a>
</div>
<div class="row">
  进度: <span id="progress">N/A</span>
</div>
<div class="row">
  预览: <br /><img alt="N/A" class="image" />
</div>
```

<!-- 第一行显示上传按钮 -->

<!-- 美化上传按钮 -->

<!-- 上传照片 -->

<!-- 第二行显示上传进度 -->

<!-- 上传进度 -->

<!-- 第三行显示预览图片 -->

<!-- 预览图片 -->

CSS 核心代码如下：

```
01 html { -webkit-tap-highlight-color: rgba(0, 0, 0, 0); } // 取消按键效果
02 .row { margin: 20px 0px; } // 设置每行的间距
03 .file { // 美化上传按钮
```



```

04     position: relative;                                // 方便子元素绝对定位
05     background: #D0EEFF;                                // 设置背景颜色
06     color: #1E88C7;                                    // 设置字体颜色
07     border: 1px solid #99D3F5;                          // 设置边框
08     border-radius: 4px;                                  // 设置圆角边框
09     padding: 4px 12px;                                   // 设置内边距
10     text-decoration: none;                              // 去除链接下划线
11     overflow: hidden;                                   // 隐藏子元素超宽部分
12 }
13 .file:hover {                                          // 美化上传按钮 hover 效果
14     background: #AADFFD;                                // 设置背景颜色
15     border-color: #78C3F3;                              // 设置边框颜色
16     color: #004974;                                    // 设置字体颜色
17 }
18 .file input {                                         // 设置上传文件元素
19     position: absolute;                                  // 绝对定位元素
20     left: 0px;                                          // 设置左上角的 x 坐标位置
21     top: 0px;                                           // 设置左上角的 y 坐标位置
22     opacity: 0;                                         // 设置为透明
23 }
24 .image {                                              // 设置预览图
25     display: none;                                       // 隐藏元素
26     width: 300px;                                       // 设置宽度
27     height: 300px;                                      // 设置高度
28     margin-top: 20px;                                   // 设置间距
29 }

```

实例布局中主要的难点是美化文件上传按钮。浏览器自带的文件上传按钮样式比较简陋，实际生产项目中普遍选择自定义“上传按钮”样式。通常，美化原理是将原始上传按钮覆盖在另外一个美化过的元素之上，然后将上传按钮设置为透明。这样用户实际看到的按钮外观其实是被美化的下层元素。当用户单击时，因为上传按钮在上一层，单击事件会在该元素处被触发。实例的 JavaScript 核心代码如下：

```

01 var image = document.querySelector("img");           // 获取图片预览元素
02 var progress = document.querySelector("#progress");  // 获取进度条元素
03 document.querySelector("#photo").onchange = function() { // 响应文件上传事件
04     var files = this.files;
05     if(files.length === 0) {                          // 如果没有选择文件，返回
06         return;
07     }
08     var file = this.files[0];                        // 获取选中上传的文件

```

```

09     uploadFile(file);                // 上传文件
10     readAsDataURL(file, image);      // 预览文件
11 }
12 function readAsDataURL(file, image){ // 预览文件
13     var reader = new FileReader();   // FileReader 用来读取文件为 Data URL 格式
14     reader.readAsDataURL(file);       // 将文件读取为 Data URL 格式
15     reader.onload = function(e){     // 响应文件读取完毕事件
16         image.src = this.result;     // 设置预览图片
17     }
18     image.style.display = "block";  // 显示预览图片元素
19 }
20 function uploadFile(file) {          // 上传文件
21     var formData = new FormData();    // 构建 Form Data
22     formData.append(file.name, file); // 将文件添加到 Form Data
23     $.ajax({                          // 用 ajax 发送文件数据
24         url: '/upload',               // 上传路径
25         type: 'POST',                // 上传方式
26         data: formData,               // 设置上传 Form Data
27         contentType: false,           // 让浏览器决定 Content Type
28         processData: false,           // 不要序列化上传的数据
29         beforeSend: function(xhr) {   // 发送 AJAX 前触发
30             xhr.upload.onprogress = function(e) { // 注册上传进度事件
31                 // 计算上传百分比
32                 var percent = Math.floor(e.loaded / e.total * 100);
33                 progress.innerHTML = percent + "%"; // 显示百分比
34             };
35         },
36         success: function(resp) {      // 注册上传成功事件
37             progress.innerHTML = "上传成功"; // 显示上传成功
38         },
39         error: function() {            // 注册上传失败事件
40             progress.innerHTML = "上传失败"; // 显示上传失败
41         }
42     });
43 }                                     // 此代码依赖 zepto.js

```

代码第 12~19 行,通过 FileReader 接口读取图片文件内容,然后赋值给 IMG 元素的 src 属性,实现图片预览功能。

代码第 20~43 行实现文件上传,通过 HTML 5 提供的 FormData 对象存储图片文件内容,并使用 AJAX 传送 FormData 实例数据至服务器。同时,通过监听 onprogress 事件实时显示上传进度。

注意: 上传进度指的是文件网络传输的进度。网络传输结束之后, 服务器还要对文件做处理, 然后响应消息, 直至客户端接收到响应消息才算文件上传完毕。

6.4.2 服务器端 Node.js

服务器端 Node.js 核心代码如下:

```
01 function handlefileUploadRequest(req, res) {           // 处理文件上传请求
02     if (req.method === "POST") {                       // 文件上传请求必须是 POST 方法
03         var form = new formidable.IncomingForm();       // 构建 Form Data 实例
04         form.on('error', function (error) {            // 监听 Form Data 上传出错事件
05             res.writeHead(500);                         // 设置错误响应头
06             res.write('上传失败, ' + error);            // 设置上传失败状态码和信息
07             res.end();                                  // 结束响应
08         }).parse(req, function(error, fields, files){    // 解析 from 上传数据
09             for(var key in files) {                     // 遍历上传文件
10                 var file = files[key];                  // 获取上传文件
11                 var img_path = path.resolve(__dirname, 'public/images', 12, file.name)
12                 var ws = fs.createWriteStream(img_path); // 构建写入流
13                 fs.createReadStream(file.path).pipe(ws); // 写入文件至服务器
14             }
15             res.writeHead(200);                          // 设置成功响应头
16             res.end();                                   // 结束响应
17         });
18     }
19 }
```

服务器的主要工作是使用 Formidable 第三方模块接收图片数据流, 将获取到的文件写入磁盘。

提示: Formidable 是 Node.js 的第三方模块, 用于在文件上传中解析表单数据。GitHub 地址为 <https://github.com/felixge/node-formidable>。

6.5 利用 Microdata 进行 SEO 优化

为了帮助机器更好地理解 Web 网页应用, 在 HTML 5 中引入了 Microdata 技术。该技术旨在让网络更加智能, 如提供信息获取、信息过滤、Web 自动服务等。

6.5.1 认识 Microdata

打个比方, 网页上出现一串数字, 这串数字到底是手机号、商品价格, 还是商品数量, 对于人脑来说, 很容易辨识出来, 但对于计算机来说却异常困难。通过引入 Microdata 技术, 希望计算机可以像人脑一样“理解”信息背后的真实含义。

Microdata 由名称和值 (Name/Value) 对组成, 以自定义的词汇表为中心, 每一个词汇表定义一组命名属性。根据词汇表, 在页面中需要在语义化的元素上添加对应的 Microdata 属性名, 这样计算机就可以正确理解元素内容的含义。

Microdata 中有一个范围 (Scoping) 的概念, 对于 Microdata 的范围, 可以理解成 DOM 元素中的父子关系。一旦在某个元素上规定了这个范围, 该元素以及子元素的所有 Microdata 属性都源于设定的词汇表。同一个页面中允许使用多个词汇表, 词汇表之间也可以内嵌, 这些从属关系都是通过对 DOM 自有结构的使用来完成。

注意: Microdata 只是用来帮助页面实现语义化, 并不是独立的数据格式, 是对 HTML 的一种补充, 并且可选。

6.5.2 提升网页 SEO 效果

HTML 网页的消费者主要有两个: Web 浏览器和搜索引擎。对于搜索引擎来说, 不但提高了网页内容的辨识度, 还会使搜索结果变得更加丰富友好, 同时提升 SEO 效果。下面通过谷歌搜索的一个例子来了解搜索引擎是如何使用 Microdata 技术的。

通过 Google 搜索关键词“Anna’s Pizzeria”, 会看到有几条搜索结果除了简单的内容摘录片段外, 还有结构化的信息显示, 如图 6.8 所示。

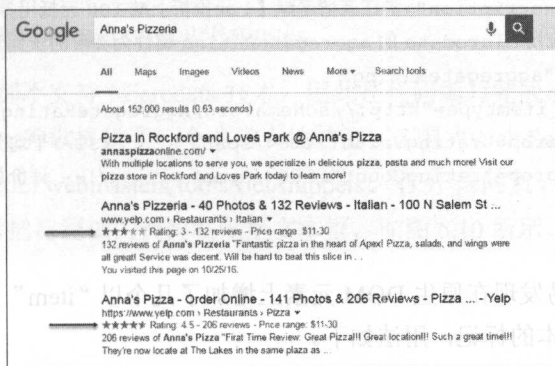


图 6.8 Google 搜索结果展示

搜索结果中的这些评分是如何得来的呢? Google 怎么会知道网页上哪些数据是评分, 哪些数据是价格? 知道了 Microdata 技术的存在, 再来看这个问题就很简单了。显示评分的网站对 HTML 内容做了语义化处理, 通过查看对应的网页源码, 可以看到网站在这些展示评分的 DOM 元素上添加了 Microdata 属性, 通过 Microdata 属性名称告诉搜索引擎这些内容的含义。Yelp 网站(国外的大众点评网)店铺详情页使用 Microdata 优化的 HTML 源码如图 6.9 所示。

```

605
606
607 <div class="biz-page-header clearfix">
608   <div class="biz-page-header-left claim-status">
609     <div class="hidden">
610       <div itemprop="aggregateRating" itemscope itemtype="http://schema.org/AggregateRating">
611         <meta itemprop="ratingValue" content="3.0">
612         <span itemprop="reviewCount">132</span>
613       </div>
614     </div>
615   </div>
616
617

```

图 6.9 Yelp 网站 HTML 源码截图

提示: 可复制链接 <view-source:https://www.yelp.com/biz/annas-pizzeria-apex> 到 Chrome 浏览器地址栏, 直接查看网页源码。

了解到 Microdata 在 Yelp 网站的实际应用之后, 动手实现一个在线课程商品的语义化处理案例, 代码如下:

```

01 <!--定义一个商品的容器, 并指定 Microdata 的作用范围 -->
02 <div itemscope itemtype="http://schema.org/Product">
03   <!-- 商品图片, 应用 Microdata 的 image 属性 -->
04   
05   <!-- 商品名称, 应用 Microdata 的 name 属性 -->
06   <p itemprop="name">大学英语四级口语特训班</p>
07   <!-- 商品描述, 应用 Microdata 的 description 属性 -->
08   <p itemprop="description">沪江亲情奉献【1 元抢折上减 100 元抵用券】</p>
09   <!-- 商品评价, 应用 Microdata 的 aggregateRating 属性为评价的子元素指定新范围 -->
10   <div itemprop="aggregateRating"
11     itemscope itemtype="http://schema.org/AggregateRating">
12     <span itemprop="ratingValue">4.8</span> <!-- 平均分 -->
13     <span itemprop="ratingCount">566</span> <!-- 评价总数 -->
14   </div>
15 </div>

```

从上述代码中很容易发现在原生 DOM 元素上增加了几个以“item”开头的属性, 这些也正是 Microdata 最常用、最基本的标记, 用法如下。

- **itemscope**: 声明词汇表的范围, 也就是 Microdata 项的开始。

- **itemtype**: 声明词汇表的命名空间, 定义一类名称/值对。
- **itemprop**: 指定数据项属性, 一般情况下属性名称的值就是元素标签里的文本值。对于有 URL 属性的元素, 该值为 URL 自身, 这些元素包含有 A、Area、Audio、Embed、IFRAME、IMG、Link、Object、Source、Video。还有例外的, 比如对于 Time 元素, 该值取自 **datetime** 属性; 对于 Meta 元素, 该值取自 **content** 属性。

熟悉了 Microdata 这几个常用属性后, 再回头来分析上文的实例代码, 就比较容易理解了。

代码第 02 行, 在最外层 DIV 元素上添加了 **itemscope** 属性, 标明 Microdata 数据项开始, 并通过 **itemtype** 属性指定该数据项使用的词汇表命名空间。对于此课程商品的实例, 采用的命名空间是 <http://schema.org/Product>, 在该命名空间下定义商品相关的属性。

提示: 如果想了解更多 Product 命名空间的信息, 可以直接打开该空间链接 <http://schema.org/Product> 查看。

代码第 04~08 行, 每个元素上都添加了 **itemprop** 属性, 用来标明该元素的数据名称, 比如 IMG 元素的数据属于空间 Product 的 **image** 属性。其中, 第 1 个 P 元素的数据属于空间 Product 的 **name** 属性, 第 2 个 P 元素的数据属于空间 Product 的 **description** 属性。

代码第 10 行, DIV 元素上添加 **itemprop** 属性的同时, 也添加了 **itemscope** 和 **itemtype** 属性。这里的 **itemprop** 跟之前的元素没有任何区别, 用于标明该项数据是 **aggregateRating** 类别。额外的 **itemscope** 和 **itemtype** 的意思等同于最上层的 DIV 元素, 表示标签下所有的子元素使用 **itemtype** 指定的命名空间。

代码第 12~13 行, 这 2 个 Span 元素里的 **ratingValue** 和 **ratingCount** 属性来自于父级元素声明的新的命名空间 <http://schema.org/AggregateRating>。

目前的 Web 浏览器都不支持 Microdata 技术, 因此在代码编写完成后, 需要工具来检测代码的正确性。针对 Microdata 谷歌提供了一个非常人性的测试工具 “Rich Snippets Testing Tool”, 网址为 <http://www.google.com/webmasters/tools/richsnippets>。打开该网页, 在弹窗中选择 “CODE SNIPPET” 选项卡, 然后把待测试的代码粘贴至文本框, 如图 6.10 所示。

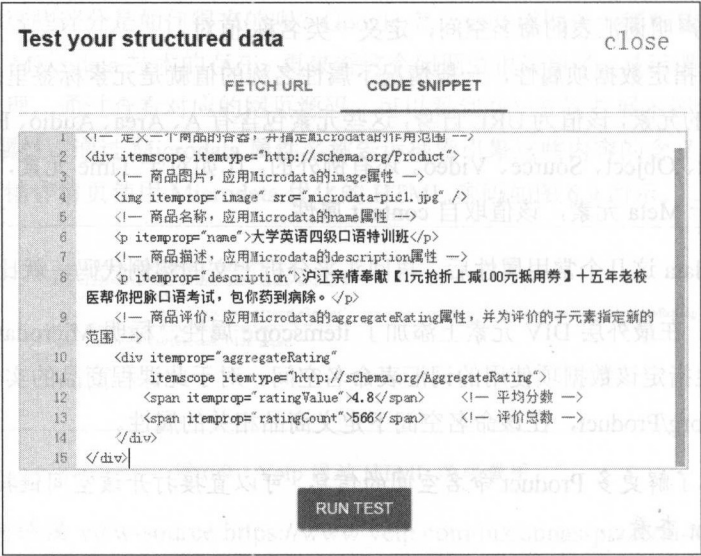


图 6.10 使用谷歌工具测试 Microdata 代码

单击“RUN TEST”按钮，可以看到代码的解析结果，如图 6.11 所示。

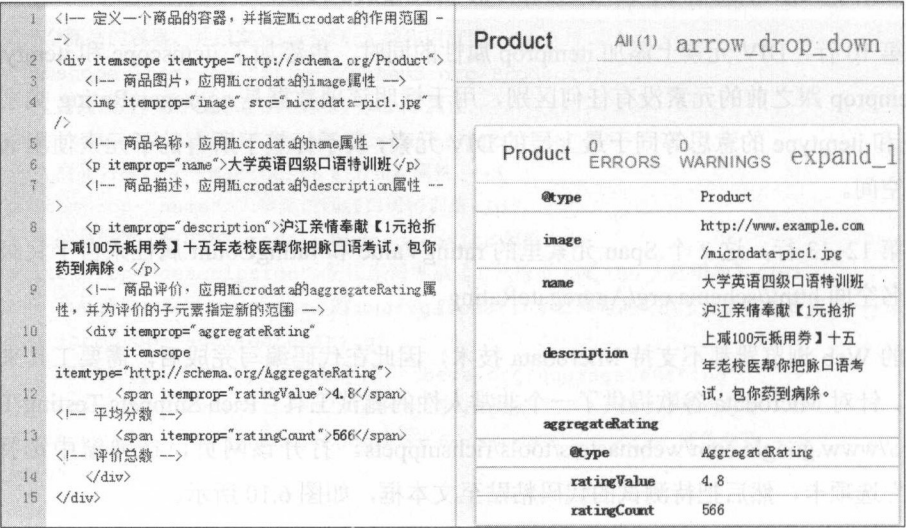


图 6.11 Microdata 测试代码的解析结果

到目前为止，读者最迷惑的恐怕就是命名空间这个概念了。在实际开发中，选择命名空间和市场上有哪些可用命名空间，是一个头疼的问题。命名空间，也就是词汇表，目前通用的词汇表

主要分为9大类,每一类都有多个提供者可选。对于这些类型和提供者的明细,见表6.1。

表 6.1 词汇表

类 型	提 供 者
事件(Events)	http://schema.org/Event
	http://data-vocabulary.org/Event
	http://www.whatwg.org/specs/web-apps/current-work/multipage/links.html#vevent
人物(Person)	http://schema.org/Person
	http://microformats.org/profile/hcard
	http://data-vocabulary.org/Person
组织或业务(Organisation or business)	http://schema.org/Organization
	http://microformats.org/profile/hcard
	http://data-vocabulary.org/Organization
日历(Calendar)	http://schema.org/Event
	http://microformats.org/profile/hcalendar#vevent
	http://data-vocabulary.org/Event
预览(Review)	http://schema.org/Review
	http://schema.org/AggregateRating
	http://microformats.org/wiki/hreview
	http://data-vocabulary.org/Review
	http://data-vocabulary.org/Review-aggregate
许可证(License)	http://n.whatwg.org/work
产品和服务(Products and services)	http://schema.org/Product
	http://microformats.org/wiki/hproduct
	http://purl.org/goodrelations
	http://data-vocabulary.org/Product
Atom提要(Atom feed)	http://microformats.org/wiki/hatom
食谱(Recipes)	http://schema.org/Recipe
	http://microformats.org/wiki/hrecipe
	http://data-vocabulary.org/Recipe

注意: Microdata 的词汇表并非不可扩展,开发者可以使用定义好的通用规范,常见的有 schema.org、data-vocabulary.org、microformats.org 等,还可以由开发者自己定义。

6.6 制作一个带字幕的视频播放器

Web 开发中,在网页中嵌入一段视频已司空见惯。在 HTML 5 大行其道的时代,传统的 Flash 播放器显然已经成为历史,如何利用 HTML 5 新技术开发一款属于自己的视频播放器,是每个

Web 前端开发者需要关注和学习的课程。本节将通过一个具体实例，讲解如何开发一款 HTML 5 的播放器。

实例播放器包含 3 个常用功能按钮：播放、快进和静音，并且配备一段自定义的视频字幕，运行效果如图 6.12 所示。



图 6.12 视频播放器运行效果

实例的 HTML 结构代码如下：

```
01 <!-- 定义一个video，用于播放视频 -->
02 <video width="600" poster="coverimg.jpg">
03   <!-- 指定视频格式和视频地址 -->
04   <source type="video/mp4" src="video_test.mp4" />
05   <!-- 指定视频字幕 -->
06   <track kind="subtitles" src="webvtt.vtt" default />
07 </video>
08 <!-- 定义按钮容器和 3 个控制按钮 -->
09 <div class="ctl">
10   <input type="button" value="播放" />
11   <input type="button" value="快进(x1)" />
12   <input type="button" value="静音" />
13 </div>
```

代码第 02 行定义了一个 Video 标签，并指定元素宽度为 600 像素。Video 是 HTML 5 规范添加的新标签，用于播放视频文件。Video 的 poster 属性用于指定该视频的预览图，设置图片的 URL 属性，可以是站内图片或站外图片。

代码第 04 行通过 Video 的子元素 Source 来设置视频的 URL 属性。实例中的视频地址为

“video_test.mp4”，并指定视频类型为“video/mp4”。

代码第 06 行 Track 标签同样是 HTML 5 中新增的特性，只作为 Video 或 Audio 等媒体元素的子元素出现，用于提供显示字幕文件功能。属性 kind 表示文本文件类型，可选值有 captions、chapters、descriptions、metadata 和 subtitles。其中本例设置的 subtitles 代表视频字幕文件。Track 标签的属性 src 指向一个纯文本文件。本实例中的“webvtt.vtt”是自定义的字幕内容，播放视频时，文件中的字幕将会按照设定的时间展示在视频上。代码中的 default 表示这是一个默认的字幕资源，视频播放时将采用这个字幕文件渲染。

代码第 10~12 行，分别定义了播放器的基本功能按钮：播放、快进和静音。

完成播放器的主体结构之后，开始通过编写 JavaScript 代码实现播放器按钮功能，实例中的功能实现代码如下：

```
01 // 获取播放器元素
02 var vd = document.querySelector('video');
03 // 获取播放器的所有控制按钮
04 var btns = document.querySelectorAll('.ctl input');
05 // 为"播放"按钮添加单击事件
06 btns[0].onclick = function () {
07     if (vd.paused) {
08         vd.play();
09         this.value = '暂停';
10     } else {
11         vd.pause();
12         this.value = '播放';
13     }
14 };
15 // 为"快进"按钮添加单击事件
16 btns[1].onclick = function () {
17     if (vd.playbackRate >= 8) vd.playbackRate = 1;
18     else vd.playbackRate *= 2;
19     this.value = '快进(' + vd.playbackRate + ')';
20 };
21 // 为"静音"按钮添加单击事件
22 btns[2].onclick = function () {
23     vd.muted = !vd.muted;
24     this.value = vd.muted ? '取消静音' : '静音';
25 };
```

代码第 02 行，通过获取 Video 元素拿到视频的实例对象，所有对播放器的操作都将通过该对

象来完成。

代码第 06~14 行, 监听播放按钮 Click 事件, 控制视频的播放和暂停。事件代码中首先通过视频的 `paused` 属性拿到视频的状态 (暂停或者播放中), 然后根据视频当前不同的状态执行 `play` (播放) 或 `pause` (暂停) 方法。

代码第 16~20 行, 监听快进按钮 Click 事件, 通过改变视频对象的 `playbackRate` 属性控制视频播放的速度, 同时设定最高速度为 8 倍原速, 当超过 8 倍速度时自动还原。

代码第 22~25 行, 监听静音按钮 Click 事件, 通过改变视频对象的 `muted` 属性控制视频是否静音。`muted` 属性为布尔类型, 值为 `true` 表示静音, 值为 `false` 表示取消静音。

除了实例中使用 Video 提供的方法、属性改变播放器的行为外, Video 还提供了对播放器行为的一些监听事件, 可以通过这些监听事件来获取播放器当前的行为和状态, 进而实现其他操作。Video 元素事件见表 6.2。

表 6.2 Video 元素事件

事件名称	功能
abort	当视频加载被异常终止时产生该事件
canplay	当浏览器可以开始播放该音视频时产生该事件
canplaythrough	当浏览器可以开始播放该音视频到结束而无须因缓冲而停止时产生该事件
durationchange	当媒体的总时长改变时产生该事件
emptied	当前播放列表为空时产生该事件
ended	当前播放列表结束时产生该事件
error	当加载媒体发生错误时产生该事件
loadeddata	当加载媒体数据时产生该事件
loadedmetadata	当收到总时长、分辨率和字轨等 metadata 时产生该事件
loadstart	当开始查找媒体数据时产生该事件
pause	当媒体暂停时产生该事件
play	当媒体播放时产生该事件
playing	当媒体从因缓冲而引起的暂停和停止恢复到播放时产生该事件
progress	当获取到媒体数据时产生该事件
ratechange	当播放倍数改变时产生该事件
seeked	当用户完成跳转时产生该事件
seeking	当用户正执行跳转时操作的时候产生该事件
stalled	当试图获取媒体数据, 但数据还不可用时产生该事件
suspend	当获取不到数据时产生该事件
timeupdate	当前播放位置发生改变时产生该事件
volumechange	当前音量发生改变时产生该事件
waiting	当视频因缓冲下一帧而停止时产生该事件

想必读者已经注意到了，实例中除了实现播放器基本的控制功能外，还包含一个自定义字幕实现。现在就来看看字幕文件的庐山真面目，文件内容代码如下：

```
01 WEBVTT
02
03 Cue-1
04 00:00:01.000 --> 00:00:05.000
05 这里设置视频 1 秒-5 秒的字幕
06
07 Cue-2
08 00:00:06.000 --> 00:00:10.000
09 这里设置视频 6 秒-10 秒的字幕
```

这是一个简洁的 WebVTT 格式文件，全称 Web Video Text Tracks，中文意思是网络视频文本轨道。WebVTT 与 HTML 5 的 Track 标签配合，可给音频或视频等媒体资源添加字幕、标题和其他描述信息。WebVTT 文件通常以“WEBVTT”开头（也可以添加字节顺序标记），紧接着是一个空行，然后是文件主题内容。WebVTT 文件的主题内容被称为“Cues”，一个 WebVTT 文件包含一个或多个“Cue”，Cue 的语法如下：

```
01 [idstring]
02 [hh:]mm:ss.msmsms --> [hh:]mm:ss.msmsms
03 Text string
```

每个 Cue 都有唯一的 ID，ID 为可选。对于时间的设置，必须严格遵守语法规则，时分秒都是两位数字，不足的前边加“0”填补，毫秒必须是三位数字。代码第 03 行是内容部分，内容可以是一行，也可以是多行。

了解了 WebVTT 的语法之后，再来看实例中的字幕文件内容，就比较容易理解了。文件分别在视频的第 1~5 秒和第 6~10 秒，定义了两段字幕内容。

WebVTT 还允许为字幕设置一些简单样式，如字号、颜色、位置等，实例中字幕样式代码如下：

```
/* 设置字幕样式：透明、黑色、阴影、40 像素大 */
video::cue {
    background:transparent;
    color: black;
    text-shadow:1px 1px 2px #fff;
    font-size:40px;
}
```


提示：有关 WebVTT 的更多知识，请访问 <http://dev.w3.org/html5/webvtt/>。

6.7 使用 Pixi.js 制作“抓住开学君”游戏（Canvas+WebGL）

Pixi.js 是一款高效并且开源的 2D 渲染引擎，主打支持硬件 GPU 渲染的 WebGL API。如果浏览器不支持 WebGL，则自动退回使用 Canvas 技术。Pixi.js 可用于开发交互图形、动画和游戏等“富视觉”应用。开发者无须专门学习 WebGL 就能感受到强大的硬件加速能力。官网地址为 <http://www.pixijs.com>。

本节主要使用 Pixi.js 开发一个“抓住开学君”的游戏，这类小游戏一般用于移动端的活动宣传页面，通过丰富有趣的交互方式使用户在页面停留更长的时间从而提升转化率。

首先来看下“抓住开学君”是怎么样的一个游戏。简单来说，就是在一个固定区域内，会随机出现名叫“开学君”的卡通人物，在人物出现时可对其进行单击，每次单击并且命中能获得 1 个积分。游戏界面如图 6.13 所示。



图 6.13 “抓住开学君”游戏截图

从图 6.13 可以看到，游戏界面包含了右上角的当前得分、“开学君”卡通人物以及“开始游戏”按钮。单击“开始游戏”按钮，“开学君”会在屏幕区域的随机位置出现，每次单击“开学

君”，右上角的分数就会加1分。首先来看下项目的目录结构，如图6.14所示。

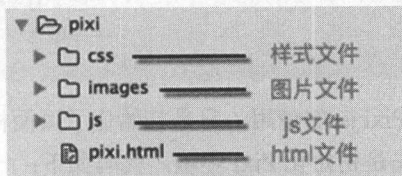


图 6.14 项目目录结构

css 文件夹中存放整个页面的样式表文件 style.css，images 文件夹存放项目中所有用到的图片文件，js 文件夹存放公共库 Pixi.js 和业务代码 main.js，pixi.html 为游戏的入口页面。pixi.html 的代码如下：

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     /*让 viewport 的宽度等于物理设备上的真实分辨率，不允许用户缩放*/
06     <meta name="viewport" content="width=device-width,initial-scale=1,user-scalable=no"/>
07     <title>抓住开学君</title>
08     <link href="./css/style.css" rel="stylesheet" type="text/css"> /* 引入 css 文件 */
09 </head>
10 <body>
11     <a class="play" href="#">开始游戏</a> /* 开始游戏按钮 */
12     <script src="./js/pixi.min.js" type="text/javascript"></script> /* 引入 pixi.js */
13     <script src="./js/main.js" type="text/javascript"></script> /* 引入业务 js */
14 </body>
15 </html>
```

接下来在 style.css 中设置一些初始化的样式和按钮的样式，代码如下：

```
01 * {
02     margin: 0; /* 初始化所有元素的外边距为 0 */
03 }
04 .play{
05     display: block; /* 把 a 标签转化为块级元素 */
06     background-color: #fe8abb; /* 背景颜色 */
07     border-radius: 10px; /* 设置圆角 */
08     text-align: center; /* 文字居中 */
09     width: 200px; /* 按钮宽度 */
10     height: 50px; /* 按钮高度 */
11     font-size: 30px; /* 字体大小 */
12 }
```

```

12     line-height: 50px;                /* 行高 */
13     text-decoration: none;            /* 去除 a 标签默认的下划线 */
14     color: #fff;                      /* 字体颜色 */
15 }

```

到现在为止, 还没有涉及 Pixi.js 的使用, 只是初始化了页面样式和按钮的样式。接下来开始制作游戏的主体逻辑部分。首先是初始化 Pixi 环境, 代码如下:

```

var result = 0,                // 得分初始化
    delay = 50,                // "开学君"出现的频率
    startPeople,               // "开学君"初始化
    explosion;                 // MovieClip 实例对象
var playBtn = document.querySelector('.play'); // 获取开始按钮的 DOM 对象
// 通过自动选择的方式创建画布 (800x600), 并设置背景为透明
var renderer = PIXI.autoDetectRenderer(800, 600, {transparent: true});
document.body.appendChild(renderer.view); // 将画布添加至容器
var stage = new PIXI.Container(); // 创建舞台容器,

```

上面代码中提到的 MovieClip 实例对象用于制作由一系列结构组成的动画, 因为“开学君”有站立和下蹲的两个动作, 实现上由两张结构图片组成, 如图 6.15 所示。

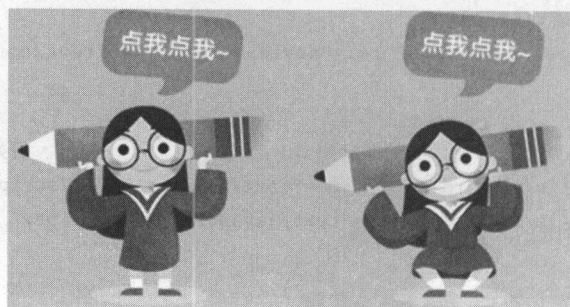


图 6.15 “开学君”的两个状态

初始化好 Pixi 的环境后, 开始在舞台上添加素材和创建动画元素, 代码如下:

```

01 PIXI.loader                // loader 方法预加载资源
02     .add('scene', 'images/game_area.png') // 加载背景图片
03     .add('man', 'images/game_begin_people.png') // 加载起始"开学君"
04     .add('images/people.json') // 加载动画"开学君"
05     .load(function(loader, res){ // 资源加载完成之后执行 load 方法
06         // 舞台中加入场景
07         var scene = new PIXI.Sprite(res.scene.texture) // 背景精灵
08         stage.addChild(scene); // 背景精灵添加到舞台

```



```

09      // 舞台中启动人物
10      startPeople = new PIXI.Sprite(res.man.texture); // 人物精灵
11      startPeople.position.set(800 / 2, 600 / 2 - 30); // 设置起始位置
12      startPeople.anchor.set(0.5); // 设置人物的起点是图片中心
13      stage.addChild(startPeople); // 人物精灵添加到舞台上
14      // 舞台中初始化分数
15      var scoreStyle = { // 分数的样式
16          fontSize: '32px',
17          fontStyle: 'normal',
18          fill: '#ff973e'
19      };
20      var score = new PIXI.Text(result + '分', scoreStyle); // 分数为文字素材
21      score.position.set(554, 20); // 初始化位置
22      score.anchor.set(0); // 设置分数的起点是左上角
23      stage.addChild(score); // 分数添加到舞台上
24      // 开学君
25      var frames = []; // 开学君的两个动作的数组
26      for(var i = 1; i < 3; i++) {
27          frames.push(PIXI.Texture.fromFrame('people-' + i + '.png'));
28      }
29      explosion = new PIXI.extras.MovieClip(frames); // 影片剪辑对象
30      explosion.anchor.set(0.5); // 起始位置为中心
31      explosion.position.set(800 / 2, 600 / 2); // 初始位置
32      explosion.animationSpeed = 0.05; // 动画速度
33      explosion.interactive = true; // 表示有事件绑定
34      explosion.on('mousedown', onDown); // 绑定鼠标单击
35      explosion.on('touchstart', onDown); // 绑定触摸单击
36      function onDown () { // 事件绑定的回调
37          result++; // 分数计算
38          score.setText(result + '分', scoreStyle);
39          // 计算开学君下次出现的位置
40          var x = Math.random() * (800 - explosion.width) + explosion.width / 2;
41          var y = Math.random() * (600 - explosion.height) + explosion.height / 2;
42          // 每次单击后开学君的位置要变换
43          explosion.position.set(x, y);
44          // 单击后开学君下次出现的时间也要刷新
45          delay = 50
46      }
47      renderer.render(stage);
48  });

```

代码第 02~04 行用于加载动画所需的素材。第 05 行的 load 方法在素材加载完毕后执行。从

上述代码中可以看出, 创建一个 PIXI 的实例并展示通常需要 6 步。

1. 创建一个舞台

舞台处于整个树形展示结构的最底层, 可以理解为背景, 所有要渲染的对象都必须连接到舞台中才能被显示。

2. 创建一个画布

画布选用 Canvas 或 WebGL 进行渲染。autoDetectRenderer 方法会自动判断当前环境是否支持 WebGL, 如果不支持则会退回使用 Canvas。

3. 将画布元素添加至页面 DOM 结构中

4. 创建一个精灵

精灵是可以直接用于舞台显示的对象, 可以理解为 DOM 中的 Element (元素节点)。精灵可以直接使用图片创建, 也可以先通过创建纹理, 再通过纹理创建精灵。纹理 (Texture) 可以理解成一种包装图片的结构, 本身不能直接用于显示, 需要借助于精灵。

5. 把精灵加入画布

6. 把画布加入舞台

到现在为止, “开学君” 游戏的素材都已经准备好, 此时的 “开学君” 还是一个静态的页面, 要让 “开学君” 动起来, 需要借助于 HTML 5 的 requestAnimationFrame 方法。

在过去, JavaScript 制作动画一般使用 setTimeout 或者 setInterval 函数实现, 大部分浏览器的显示频率是 16.7 毫秒 (即每秒显示 60 帧)。但实际情况是开发者无法知晓浏览器运行时的渲染更新频率, 因为可能在两段间隔之间发生了其他绘制动作导致执行被堵塞, 这也是为什么使用 setTimeout 或者 setInterval 函数实现动画会出现卡顿, 即常说的掉帧。

为了有效地解决动画卡顿的问题, HTML 5 重新提供了一个新的方法 requestAnimationFrame, 让浏览器自身去分配渲染执行而不是人为的干预。在 “抓住开学君” 的游戏中, 运用了 requestAnimationFrame 这个方法保持画布的重绘, 实例代码如下:

```
01 function animate() {
02     // 变动位置
03     delay--; // 延时变动开学君在舞台中出现的位置
04     while(delay === 0){
05         // 计算开学君在舞台中随机出现的位置
06         var x = Math.random() * (800 - explosion.width) + explosion.width / 2;
07         var y = Math.random() * (600 - explosion.height) + explosion.height / 2;
```

```

08      // 设置开学君在舞台中出现的位置
09      explosion.position.set(x, y);
10      // 开学君出现在新的位置后刷新延迟值
11      delay = 50
12  }
13  // 递归执行 animate 方法, 是画布不断重绘
14  requestAnimationFrame(animate);
15  renderer.render(stage);
16  }

```

为了控制游戏开始时间, 需要让用户来决定 `animate` 函数的第一次执行时间, 所以在“开始游戏”的按钮上监听 Click 事件, 用于 `animate` 函数的初次执行, 代码如下:

```

var playBtn = document.querySelector('.play');          // 获取开始按钮的 DOM 对象
playBtn.addEventListener('click',function(){
    this.style.display = 'none';                        // 游戏开始后隐藏开始按钮
    stage.removeChild(startPeople);                    // 游戏开始隐藏初始开学君
    stage.addChild(explosion);                          // 舞台中插入影片剪辑的实例对象
    explosion.play();                                  // 影片剪辑开始执行自身动画
    animate();                                          // 第一次执行 animate 方法
});

```

到现在为止, 整个“抓住开学君”游戏的开发已经完成。由于 PIXI 对象的 `loader` 方法调用 `XMLHttpRequest` 进行请求, 浏览器对跨域请求有安全限制, 所以如果想测试这个游戏需要在本地搭建一个简单的 HTTP 服务器。比如 Mac 系统自带了 Apache, 可以把 PIXI 整个项目放到 Apache 服务器的文件路径 (`/Library/WebServer/Documents`) 下, 然后打开终端启动服务, 命令如下:

```
sudo apachectl start
```

在 Chrome 浏览器中访问 `http://127.0.0.1/pixi/pixi.html`, 就能测试“抓住开学君”的游戏。

如果是 Windows 系统, 可以下载 XAMPP, 快速启动一个本地服务, 具体可以参考 <http://www.apachefriends.org/>。

游戏运行效果如图 6.16 所示。

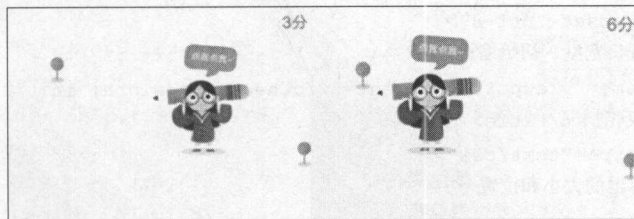


图 6.16 “抓住开学君”运行时截图

这个实例只实现了“抓住开学君”的核心功能,要真正呈现给用户还得进行一些优化和润色,比如可以加入倒计时、游戏可玩次数和好友成绩 PK 等,有兴趣的读者可以自行尝试实现。

6.8 用 Canvas 制作刮刮卡

逢年过节在热闹的集市上经常有推车卖抽奖刮刮卡,而在互联网上也有模拟现实生活中刮刮卡的应用,非常典型的是某付宝每次付款成功之后会有一次刮刮卡的抽奖。实现刮刮卡的效果有多种方式,本节主要给大家演示使用 Canvas 在 Web 页面中的实现。

Canvas 是 HTML 5 的新增元素,用于图形的绘制,如绘制路径、盒、圆、字符以及添加图像。

首先刮刮卡效果分三种状态,如图 6.17 所示。



图 6.17 刮刮卡游戏的三种状态

从图 6.17 可以看出,刮刮卡游戏的三个状态分别为初始状态、未中奖和中奖。在初始状态下,按住鼠标并拖动,会随机出现中奖或者未中奖的状态。其原理是在中奖或者未中奖两张图片中随机出现一张,然后在图片的上方用 Canvas 绘制一个灰色蒙层,同时监听鼠标或者手势移动来绘制一个透明图形,这样就能看到蒙层下方真实的图片,从而实现刮刮卡的效果。

要制作刮刮卡游戏,第一步是在 HTML 中加入 Canvas 元素,并设置对应父元素的位置,HTML 文件的代码如下:

```
01 <!DOCTYPE HTML>
02 <html>
03   <head>
04     <meta charset="utf-8">
05     /* 设置视口宽度、初始缩放 */
06     <meta name="viewport" content="width=device-width; initial-scale=1.0">
07     <title>刮刮卡</title>
08     <style type="text/css">
09       /* 设置容器的大小和位置 */
10       .container{
11         width:320px;
```

```

12         margin:10px auto 20px auto;
13         min-height:300px;
14     }
15 </style>
16 </head>
17 <body>
18     <div class="container">
19         <canvas></canvas>
20     </div>
21 </body>
22 </html>

```

为了提升用户体验,在拖动时禁止选择文本行为,通过禁用鼠标选择事件实现,代码如下:

```

var bodyStyle = document.body.style;
bodyStyle.mozUserSelect = 'none';
bodyStyle.webkitUserSelect = 'none';

```

开始定义图片类,获取 Canvas 元素,并设置背景和位置属性。在本例中用到两张随机照片,每次刷新随机一张图片作为背景,代码如下:

```

var img = new Image(); // 实例化一个图片类
var canvas = document.querySelector('canvas'); // 拿到 canvas 的 DOM 对象
canvas.style.backgroundColor='transparent'; // canvas 的背景为透明
canvas.style.position = 'absolute'; // canvas 的定位方式为绝对定位
var imgs = ['bg_01.jpg', 'bg_02.jpg']; // 中奖和未中奖两个图片
var num = Math.floor(Math.random()*2); // 随机生成 0 或 1
img.src = imgs[num]; // 图片的实例对象设置图像 url

```

进入主体部分,等待图片加载完成后,定义初始化的属性和事件函数,蒙层通过 layer 方法绘制。当按下鼠标或者单击手势时,获取当前坐标信息,并绘制透明小圆点,代码如下:

```

01 img.addEventListener('load', function(e) {
02     var ctx;
03     var w = img.width, // 获取图片宽度
04         h = img.height; // 获取图片高度
05     var offsetX = canvas.offsetLeft, // 获取 canvas 相对于左边界的偏移
06         offsetY = canvas.offsetTop; // 获取 canvas 相对于上边界的偏移
07     var mousedown = false; // 防止手势操作滑出手机边界
08     function layer(ctx) { // 绘制蒙层
09         ctx.fillStyle = 'gray'; // 蒙层颜色
10         ctx.fillRect(0, 0, w, h); // 蒙层的位置和大小
11     }

```



```

12     function eventDown(e) {                                // 鼠标或者手势按下时触发的事件回调
13         e.preventDefault();
14         mousedown = true;
15     }
16     function eventUp(e) {                                    // 鼠标或者手势松开时触发的事件回调
17         e.preventDefault();
18         mousedown = false;
19     }
20     function eventMove(e) {                                  // 鼠标或者手势移动时触发的事件回调
21         e.preventDefault();
22         if(mousedown) {
23             // 拿到 TouchList 对象中最后一个事件对象
24             if(e.changedTouches){
25                 e = e.changedTouches[e.changedTouches.length-1];
26             }
27             // 获得当前鼠标或者手势的坐标
28             var x = (e.clientX + document.body.scrollLeft || e.pageX) - offsetX || 0,
29                 y = (e.clientY + document.body.scrollTop || e.pageY) - offsetY || 0;
30             ctx.beginPath()                                  // 创建一个新的路径
31             ctx.arc(x, y, 20, 0, Math.PI * 2);              // 绘制弧线
32             ctx.fill();                                       // 填充路径
33         }
34     }
35 });

```

最后, 通过 Canvas 绘制图形, 并且监听手势及鼠标事件, 代码如下所示:

```

img.addEventListener('load', function(e) {
    // 接上段代码
    canvas.width = w;                                         // 设置宽度
    canvas.height = h;                                       // 设置高度
    canvas.style.backgroundImage = 'url('+img.src+')';        // 设置背景
    ctx = canvas.getContext('2d');                           // 返回 canvas 的上下文
    ctx.fillStyle = 'transparent';                           // 填充背景色为透明
    ctx.fillRect(0, 0, w, h);                                // 填充的位置和长宽
    layer(ctx);                                                // 绘制蒙层
    ctx.globalCompositeOperation = 'destination-out';        // 路径与原图像重叠部分透明
    canvas.addEventListener('touchstart', eventDown);
    canvas.addEventListener('touchend', eventUp);
    canvas.addEventListener('touchmove', eventMove);
    canvas.addEventListener('mousedown', eventDown);
    canvas.addEventListener('mouseup', eventUp);

```

```
canvas.addEventListener('mousemove', eventMove);
});
```

使用 Chrome 浏览器打开 index.html 文件, 体验已经完成的刮刮卡游戏。读者还可以把背景图片做得更美观, 并且配合后端服务器做成一个真正的刮刮卡应用。

6.9 实战演练: 实现 3D 全景效果

3D 全景效果能给用户带来强烈的视觉冲击, 近些年在 Web 端产品中变得异常火热, 随着 HTML 5 和 CSS 3 在现代浏览器中的普及, 开发者完全可以用新特性实现这类场景。

6.9.1 需要的 CSS 3 特性

Transform: 对元素应用 2D 或 3D 转换。该属性允许对元素进行旋转、缩放、移动或倾斜, 语法如下:

```
transform: none | transform-functions
```

- **translate3d(x,y,z):** 通过一个三维向量表达一个元素在 3D 空间内的移动距离。
- **rotate3d(x,y,z,a):** 通过一组矢量[x,y,z]和旋转角度描述 3D 旋转。

Transform-Origin: 改变被转换元素的位置, 该属性必须与 Transform 属性一同使用, 语法如下:

```
transform-origin: x-axis y-axis z-axis;
```

Transform-Style: 固定如何在 3D 空间中呈现被嵌套的元素, 语法如下:

```
transform-style: flat|preserve-3d;
```

Perspective: 定义 3D 元素距视图的距离, 以像素计, 语法如下:

```
perspective: number | none;
```

6.9.2 实现原理

3D 全景效果的实现首先需要理解在计算机世界所有的东西都是离散的。比如计算机要在屏幕中渲染一个圆形, 其实是由一个有足够多边的正多边形绘制而成, 理解 3D 全景效果也一样。开发者需要使用多张图片通过 Transform 属性在空间中进行排列, 给用户一种连续“全景”图片的感受。

在真正实现效果前, 先需要梳理一下 CSS 3 特性在 3D 全景效果中的应用, 首先是 Perspective

属性, 该属性存在与否决定了用户能否看到一个 3D 的场景。简单来说 **Perspective** 通过调整用户的位置, 将可视内容映射到一个视锥上, 继而投到一个平面, 可以想象为一个用户眼睛为顶点, 屏幕为底边的三角形, 从而提供透视能力并产生景深。

接着就可以进入图片排列的过程, 对于单张图片, 需要进行两个方向上的处理, 一方面要进行 **rotateY** 的处理, 以确保每张图片能够面向中心点, 这样相邻元素间的角度等于 360 度除以图片的数量。另一方面要将图片元素向外 (即 Z 轴方向) 平移一定的距离, 使之能够拼接完整的图形。

全景图俯视效果, 如图 6.18 所示。

开发者需要计算如图 6.18 的 r 值并设置为 **translateZ** 的值, 代码如下:

```
/* panelSize 为每面的宽度, numberOfPanels 为面的数量 */
r = Math.round( ( panelSize / 2 ) / Math.tan( Math.PI / numberOfPanels ) );
```

利用上述公式, 可以推算出图 6.18 正九边形的 r 值, 计算公式如下:

$$(210\text{px} / 2) / \tan(\text{Math.PI} / 9) = 288\text{px}$$

一般来说, 制作一副 3D 全景图最少需要 6 张图, 这并不是意味着需要一个六边形。实际上制作的效果常被称为“天空盒”, 仅需要一个四边形进行 **rotateY** 的变换和计算, 剩下两张进行 **rotateX** 和 **rotateZ** 的变换用作天空和地面, 如图 6.19 所示。

最后是 **Perspective** 值的设置, **Perspective** 值没有一个精确的数值, 需要在开发中进行调整, 距离太远会导致失真, 距离太近则会缺乏立体感。

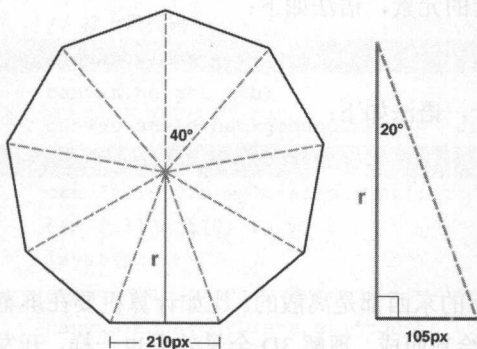


图 6.18 全景图俯视效果

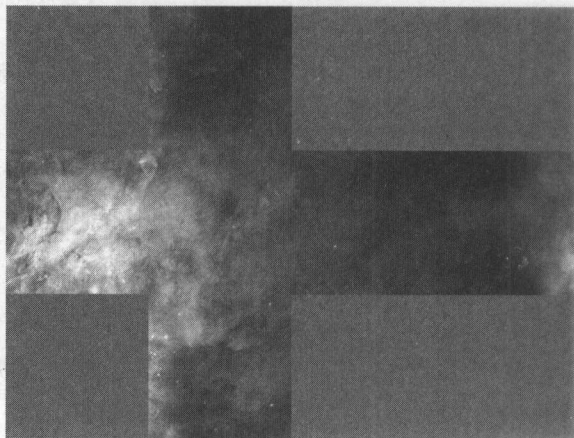


图 6.19 天空盒图制作实例

6.9.3 实现代码

HTML 代码如下:

```
<div class="scene">
  <div class="container">
    <div class="cube">
      <!-- 设置前后左右上下六个面组成天空盒 -->
      
      
      
      
      
      
    </div>
  </div>
</div>
```

CSS 代码如下:

```
.scene {
    /* 给整个场景设置 perspective */
    position: relative;
    perspective: 600px;
}
/* 增加一层设置 translateZ 方便控制"眼睛"到场景的距离
    可以通过外层的 perspective 控制, 但改变 perspective 会影响整个场景, 不推荐 */
.container {
    position: relative;
    transform-style: preserve-3d;
    transform: translateZ(300px);
}
.cube {
    /* 设置 preserve-3d 来展示 3d 透视 */
    position: relative;
    /* ...省略部分样式代码, 详见光盘源码
    transform-style: preserve-3d;
}
.cube img {
    width: 600px;          /* 图片的宽度 */
    height: 600px;         /* 图片的高度 */
}
.face {
    display: block;
    position: absolute;     /* 决定定位设置每个面的位置 */
```



```

left: -300px;
top: -200px;          /* 调整盒子的位置让观察点在盒子的内部稍低于中心点的位置 */
backface-visibility: hidden; /* 保证视线后面的图像不会渲染, 可不加 */
}
/* 调整每个面的位置, 上下面需要变换 X 和 Z 轴, 另四个面变换 Y 轴,
   因为是正方体, 角度为 90 度, 具体详见实现原理 */
.front { transform: translate3d(0, 0, 300px) rotate3d(0, 1, 0, 180deg); }
.back { transform: translate3d(0, 0, -300px); }
.left { transform: translate3d(-300px, 0, 0) rotate3d(0, 1, 0, 90deg); }
.right { transform: translate3d(300px, 0, 0) rotate3d(0, 1, 0, -90deg); }
.top { transform: translate3d(0, -300px, 0) rotate3d(1, 0, 0, -90deg) rotate3d(0, 0, 1, 90deg); }
.bottom { transform: translate3d(0, 300px, 0) rotate3d(1, 0, 0, 90deg) rotate3d(0, 0, 1, -90deg); }

```

最后需要 JavaScript 来控制整个场景的移动, 这部分是一个通用的监听整个场景手势事件的过程, 代码如下:

```

01 // 代码主要展示 touchstart, touchmove, 其他详见源码
02 var speed = 100; // 设置移动速度
03 var cube = document.getElementsByClassName('cube')[0]; // 保存 cube 的 dom
04 // 分辨存储 touchstart 的偏移值和 touchmove 的偏移值
05 var lastPosX, lastPosY, curPosX, curPosY;
06 document.addEventListener('touchstart', function(e) {
07     lastPosX = e.touches[0].pageX; // 获取初始在 X 轴的偏移
08     lastPosY = e.touches[0].pageY; // 获取初始在 Y 轴的偏移
09 })
10 document.addEventListener('touchmove', function(e) {
11     curPosX = e.touches[0].pageX;
12     curPosY = e.touches[0].pageY;
13     // 计算实际投射到 cube 的偏移角度
14     valLR = -1 * speed * (curPosX - lastPosX) / 360;
15     valTB = speed * (curPosY - lastPosY) / 360;
16     // 操作 cube 变换
17     value = 'rotateX(' + valTB + 'deg) rotateY(' + valLR + 'deg)';
18     cube.style.transform = value;
19 })

```

代码第 14 行~15 行, `pageX` 变化时, 鼠标横向偏移, 图片进行 Y 轴变换, `pageX` 增大, Y 轴角度变小。`pageY` 变化时, 鼠标纵向偏移, 图片进行 X 轴变换, `pageY` 增大, X 轴角度增大。通过拖动的距离计算旋转的角度, 如果拖动一屏为 360 度, 则通过两点的距离差计算百分比, 同时通过 `speed` 变量调整转动的速度。

6.10 本章小结

HTML 5 依托于移动互联网的迅速发展,走过了一段疯狂爆发的时期,将 Flash 技术远远甩在身后。相信,随着更多主流浏览器支持的实现,HTML 5 的潜力还将继续被挖掘,在人机交互领域展现出更多的可能。本章主要通过地理位置、手势事件、WebSocket、WebRTC、语义化、音视频、Canvas、WebGL 等技术,向读者介绍 HTML 5 在移动场景下的实战。读者不妨在阅读实例的同时,亲自动手实现,快速掌握这门在前端领域举足轻重的技术。

7

第 7 章

移动网页样式布局实战

随着手机等移动设备的普及，用户的使用场景开始大面积从 PC 端转移至移动端，Web 前端的开发工作也更多地偏向移动端应用。所以，有效地掌握移动端网页样式布局，是成为一名合格前端开发工程师的必要条件。移动端因其较好的 HTML 5 和 CSS 3 支持程度，在开发布局上可以尝试更多新的特性，而不必像 PC 时代一样需要考虑到各种版本浏览器的兼容性，尤其是 Internet Explorer 系列。当然移动端也并非完美，兼容性问题依旧存在，这也许是做前端开发的乐趣所在吧。

本章将向读者介绍移动端常用布局，如 Media Query、Flex、rem 等，帮助读者快速掌握移动端布局方法。

7.1 静态布局的实际应用

静态布局又称固定宽度布局，这种布局的特点是布局的大小不会随用户调整浏览器窗口大小

而变化。在传统 PC 端一般选择 960 像素，因为这个宽度可以被所有现代浏览器适配，并且能被主流屏幕宽度比例整除，在多栏布局中不会存在带小数的像素点。而在移动端一般选择 320 像素，这个大小正好是 960 像素的 1/3，也具有同样的优点。

7.1.1 设计活动页面静态布局

对于传统静态布局，在 PC 端上如果屏幕宽度小于设定的宽度，会出现 Y 轴滚动条提示用户进行拖动。如果屏幕宽度大于设定的宽度，则在屏幕两侧出现空白。在移动设备上这种体验会比较糟糕，不过开发者可以通过整体缩放或者媒体查询进行适配，如图 7.1 所示。



图 7.1 活动页面静态布局实例

7.1.2 静态布局在移动端上的自适应

1. 整体缩放

整体缩放可以用在营销活动页，营销活动可能因为设计美观的需求必须使用背景图片而非背景色，因此需要考虑背景图适应屏幕大小。开发者可以用 320 像素的宽度作为基础宽度（高度可以固定），然后通过计算实际文档宽度来进行相应缩放。

使用整体缩放布局开发的项目在加载过程中需要监听 `resize` 事件, 代码如下:

```
window.addEventListener('resize', document.body.clientWidth / 320);
```

除此之外, 开发者还需要在缩放的节点上添加 `transform-origin` 属性保证缩放从原点开始, 如果不是原点缩放会导致页面不能完全显示, CSS 代码如下:

```
.wrap {
  width: 320px;
  transform-origin: 0 0
}
```

2. 媒体查询

另一种方法是使用背景色整体填充, 主体使用不同的媒体查询进行大小优化, 开发者需要对主流的屏幕分辨率进行设定, 本实例因为要处理图标到背景图的精确定位, 所以并不适合使用媒体查询, 媒体查询更加适合一些展示型页面, 本实例仅对外层做修改, 假设拥有一个绿色背景, 代码如下:

```
.wrap { background-color: green; } /* 设定外层背景色 */
.main-bg {
  background: url(../images/main.jpg) no-repeat; /* 背景图片 */
  background-size: 100%; /* 背景图片撑满容器 */
}
@media (max-width: 320px) { /* 适配 320px 宽度以下屏幕 */
  .main-bg {
    width: 320px;
    height: 470px;
  }
}
@media (min-width: 321px) and (max-width: 639px) { /* 适配 320px 到 640px 宽度屏幕 */
  .main-bg {
    width: 480px; /* 采用 480px 适配该区间 */
    height: 705px; /* 调整高度放置图片拉伸 */
  }
}
@media (min-width: 640px) { /* 适配 640px 以上宽度屏幕 */
  .main-bg {
    width: 640px;
    height: 940px;
  }
}
```

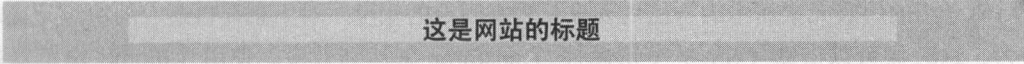
媒体查询会使代码量大幅增加,在实际开发中,开发者需要注意两点:一是将不需要根据屏幕变化的属性放到媒体查询外设置,减少代码冗余;二是设置好需要的媒体查询断点。

7.2 水平居中与垂直居中实战

水平居中和垂直居中是老生常谈的话题,也是实际开发中最常用的知识点。文本或其他行内元素的水平和垂直居中比较容易实现,但是块级元素就相对麻烦,尤其是自适应内容的块级元素。本节实战将介绍水平和垂直居中的常用方法,帮助读者在实际项目中更好地解决水平和垂直居中的问题。

7.2.1 水平居中

在 CSS 中对元素进行水平居中非常简单。如果是一个行内元素,就对其父元素设置样式 `text-align` 值为 `center`; 如果是一个块级元素,就对自身设置样式 `margin` 值为 `auto`。实现水平居中的效果,如图 7.2 所示。



这是网站的标题

图 7.2 水平居中的行内元素和块级元素

从图 7.2 可以看出,文本在方框内水平居中,整个方框在浏览器视窗内居中。因为设置了方框的宽度为 960 像素,浏览器的宽度大于 960 像素,通过样式 `margin` 值为 `"0 auto"`,使得整个方框在浏览器中水平居中,代码如下:

```
01 <!DOCTYPE html>                                <!-- 声明为 HTML5 文档 -->
02 <html>
03 <head>
04     <meta charset="UTF-8">
05     <title>水平居中</title>
06     <style type="text/css">
07         body { background-color: #ccc; } <!-- body 的背景为灰色 -->
08         /* 设置方框上下外边距为 0 并且水平居中,宽度为 960px 以及灰色背景 */
09         .container { margin: 0 auto; width: 960px; background-color: #ddd; }
10         h1 { text-align: center; }                <!-- h1 中的文本水平居中 -->
11     </style>
12 </head>
13 <body>
```

```

14     <div class="container">
15         <h1>这是网站的标题</h1>
16     </div>
17 </body>
18 </html>

```

块级元素的水平居中可以采用绝对定位的方式实现,代码如下:

```

.container {
    width: 960px;           /* 宽度为 960px */
    position: absolute;     /* 绝对定位 */
    left: 50%;              /* 左边缘向右偏移 50% 的父元素的宽度 */
    margin-left: -480px;     /* 向左移动自身宽度一半 */
}

```

采用绝对定位的方式,让元素的左边缘先移动到具有定位属性的祖先元素的正中心,然后利用负外边距将其向左移动自身宽度的一半,从而达到水平居中的效果。

7.2.2 自适应块级元素水平居中

前面介绍的块级元素居中的方法均要求元素有固定的宽度,对于想要自适应的块级元素却无能为力。这时候可以借助 CSS 3 变形属性 Transform 来达到效果。使用 Chrome 浏览器打开 transform-x.html 文件,如图 7.3 所示。



图 7.3 自适应块级元素水平居中

从图 7.3 可以看出,一共包含 4 个图标,鼠标移到每个图标上方后会显示不同的提示信息,并且提示信息相对于图标水平居中。要实现这个效果的关键是设置元素的变形属性,CSS 代码如下:

```

position: absolute;           /* 绝对定位 */
left: 50%;                   /* 左边缘移动到父元素的中心 */
transform: translateX(-50%); /* 左边缘向左移动自身宽度的一半 */

```

在 `translateX` 变形函数中使用百分比时, 是以该元素自身的宽度为基准进行换算和移动的。所以元素的宽度即使不固定, 也能实现水平居中, 达到元素自适应内容的效果。

7.2.3 行内元素垂直居中

单行文本的垂直居中只需要设置元素的高度 `height` 等于元素的行高 `line-height` 即可。单行文本垂直居中的效果如图 7.4 所示。

单行文本垂直居中的效果

图 7.4 单行文本垂直居中

要实现如图 7.4 所示单行文本垂直居中的效果, 代码如下:

```
01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     <title>垂直居中</title>
06     <style type="text/css">
07         .row {
08             width: 300px;           /* 宽度 */
09             height: 50px;           /* 高度 */
10             line-height: 50px;      /* 行高 */
11             border: 1px solid #ccc; /* 1 像素灰色实线边框 */
12             text-align: center;      /* 文本水平居中 */
13         }
14     </style>
15 </head>
16 <body>
17     <div class="row">单行文本垂直居中的效果</div>
18 </body>
19 </html>
```

从上述实例代码中可以看出, 当元素的高度和行高相等时, 其中的文本呈现垂直居中。

多行文本的垂直居中需要分两种情况, 比较简单的是不固定高度的垂直居中, 可以通过设置 `padding` 实现, 实例代码如下:

```
01 <!DOCTYPE html>
02 <html lang="en">
```



```

03 <head>
04   <meta charset="UTF-8">
05   <title>垂直居中</title>
06   <style type="text/css">
07     .row {
08       width: 200px;           /* 宽度 */
09       padding: 50px;         /* 内边距 */
10       border: 1px solid #ccc; /* 1 像素灰色实线边框 */
11     }
12   </style>
13 </head>
14 <body>
15   <div class="row">归雁划过长空，像是种下了某种不知名的思量与神往；盈月刺穿窗枢，像是植入了某种
16   不知名的思绪与惆怅</div>
17 </body>
18 </html>

```

上述代码实现的效果如图 7.5 所示。

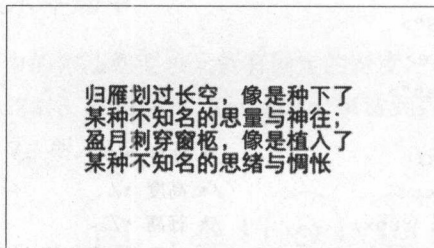


图 7.5 多行文本垂直居中

如果是固定高度的多行文本需要实现垂直居中的效果，可以使用样式 `display` 设置为 `table`，配合样式 `vertical-align` 设置为 `middle` 来实现，代码如下所示：

```

01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04   <meta charset="UTF-8">
05   <title></title>
06   <style type="text/css">
07     .wrap{
08       height: 200px;           /* 外部容器高度 */
09       display: table;         /* 此元素会作为块级表格来显示 */
10     }

```

```

11     .content{
12         vertical-align: middle;          /* 把此元素放置在父元素的中部 */
13         display: table-cell;             /* 此元素会作为一个表格单元格显示 */
14         border: 1px solid #ccc;         /* 1 像素灰色实线边框 */
15         background-color: #f0f0f0;      /* 浅灰色背景 */
16         width: 400px;                   /* 宽度 */
17     }
18 </style>
19 </head>
20 <body>
21     <div class="wrap">
22         <div class="content">秋天，是个堆满思念的季节。或许是因为坐落在这遥隔千里的
23 异国他乡，亦或是因为深居在悠长的秋日，总是会有些莫名的思绪浮出脑海。</div>
24     </div>
25 </body>
26 </html>

```

样式 `display` 设置为 `table` 其实是使元素模拟 Table 表单样式，再配合样式 `vertical-align` 设置为 `middle` 就能实现固定高度垂直居中的效果，如图 7.6 所示。

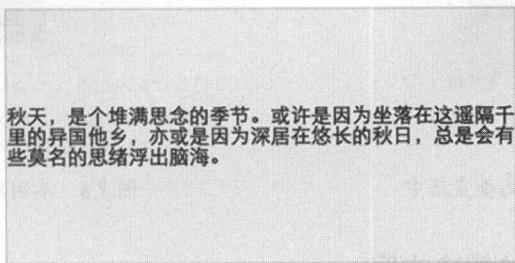


图 7.6 固定高度多行文本垂直居中

7.2.4 块级元素的垂直居中

块级元素的垂直居中也分两种情况。

一种是如果元素为固定宽高，可以使用绝对定位的方法，让元素的上边缘先移动到具有定位属性的祖先元素的正中心，然后利用负外边距将其向上移动自身高度的一半，从而达到垂直居中的效果，固定宽高垂直居中效果如图 7.7 所示。固定宽高垂直居中代码如下：

```

.container {
    width: 100px;          /* 宽度 */
    height: 100px;         /* 高度 */

```

```

position: absolute;          /* 绝对定位 */
left: 50%;                  /* 左边缘向右偏移 50% 的父元素的宽度 */
top: 50%;                   /* 上边缘向下偏移 50% 的父元素的宽度 */
margin-left: -50px;         /* 向左移动自身宽度一半 */
margin-top: -50px;          /* 向上移动自身高度一半 */
}

```

另外一种情况是不固定宽高, 需要通过 `translate` 变形函数来处理, 不固定宽高垂直居中效果如图 7.8 所示。不固定宽高垂直居中实例代码如下:

```

.container {
  position: absolute;          /* 绝对定位 */
  left: 50%;                  /* 左边缘向右偏移 50% 的父元素的宽度 */
  top: 50%;                   /* 上边缘向下偏移 50% 的父元素的宽度 */
  transform: translate(-50%, -50%); /* 左/上边缘向左/上移动自身宽/高度的一半 */
}

```

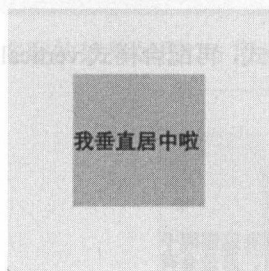


图 7.7 固定宽高垂直居中

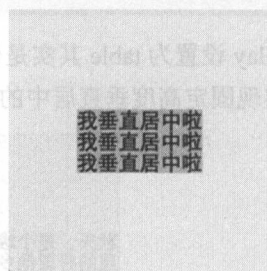


图 7.8 不固定宽高垂直居中

7.2.5 基于视口单位的解决方案

基于绝对定位的方案虽然不错, 但是对于整个布局的影响非常强烈。如果开发者不想使用绝对定位, 仍然可以采用 `translate` 把元素以其自身宽高一半为距离进行移动, 那么如何把元素左上角放置在容器的正中心呢?

如果只是想将元素相对于视口进行居中, 可以采用视口长度单位。

- 1vw 表示视口宽度的 1%。
- 1vh 表示视口高度的 1%。
- 当视口宽度小于高度时, 1vmin 等于 1vw, 否则等于 1vh。
- 当视口宽度大于高度时, 1vmax 等于 1vw, 否则等于 1vh。

采用视口长度单位实现的垂直居中效果, 如图 7.9 所示。

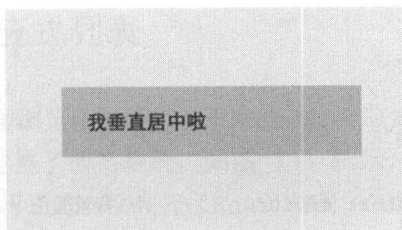


图 7.9 视口长度单位实现的垂直居中效果

实例代码如下：

```

01 <!DOCTYPE html>
02 <html lang="en">
03 <head>
04     <meta charset="UTF-8">
05     <style type="text/css">
06         body { background-color: #f0f0f0; }          /* 背景颜色 */
07         .container {
08             width: 200px;                             /* 宽度 */
09             padding: 20px;                             /* 内边距 */
10             margin: 50vh auto 0;                       /* 外边距，采用视口长度单位 */
11             transform: translateY(-50%);               /* 上边缘向上移动自身高度的一半 */
12             background-color: #ccc;                   /* 背景色 */
13         }
14     </style>
15 </head>
16 <body>
17     <div class="container">我垂直居中啦<br/></div>
18 </body>
19 </html>

```

效果虽然好，但是有较大的局限性，只能够适用于在视口居中的场景。所以日常使用中，不推荐使用该方式实现，读者可以当作一种了解。

7.2.6 基于 Flexbox 的解决方案

Flexbox 是目前推荐的最佳解决方案，专门针对这类需求所设计，但是浏览器对 Flexbox 的支持程度没有之前讨论的方案兼容程度高，但在移动端开发中完全可以使用，实例代码如下：

```

01 <!DOCTYPE html>
02 <html lang="en">

```



```
03 <head>
04 <meta charset="UTF-8">
05 <title></title>
06 <style type="text/css">
07     body {
08         background-color: #f0f0f0;      /* 背景颜色 */
09         display: flex;                  /* 弹性布局 */
10         min-height: 100vh;              /* 最小高度为 100% 视口高度 */
11         margin: 0;                      /* 外边距为 0 */
12     }
13     .container {
14         margin: auto;                    /* 外边距全为 auto */
15         padding: 20px;                  /* 内边距为 20 像素 */
16         background-color: #ccc;         /* 背景颜色 */
17     }
18 </style>
19 </head>
20 <body>
21     <div class="container">
22         我垂直居中啦<br/>
23     </div>
24 </body>
25 </html>
```

实现的效果如图 7.10 所示。

注意：当使用 Flexbox 的时候，样式 `margin` 设置为 `auto` 不仅在水平方向上居中，在垂直方向上同样居中。如果仅需实现单独的垂直居中需求，需使用 `align-self` 设置为 `center`。关注盒对齐模型的知识，读者可以访问 <http://w3.org/TR/css-align-3>。

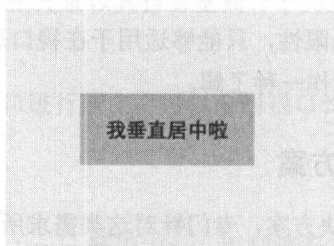


图 7.10 Flexbox 实现垂直居中

7.3 栅格系统实现响应式列表

栅格系统（布局）是从桌面浏览器时代流传下来的一种响应式布局方式，主流的 CSS 框架诸如 Blueprint、Bootstrap 等都内置了栅格系统，如图 7.11 所示。栅格系统的实现，通常将容器（Container）均分为 12 等份，再通过一系列行（Row）与列（Column）的组合创建页面布局并调整内外边距，结合 CSS 的媒体查询可以实现一个响应式的栅格系统。

注意：栅格系统的列数并没有明确的规定，也有 6 列或 24 列栅格，由于 12 列的栅格可以被 1、2、3、4、6 等分，因此具有更多的灵活性，所以通常划分为 12 列。

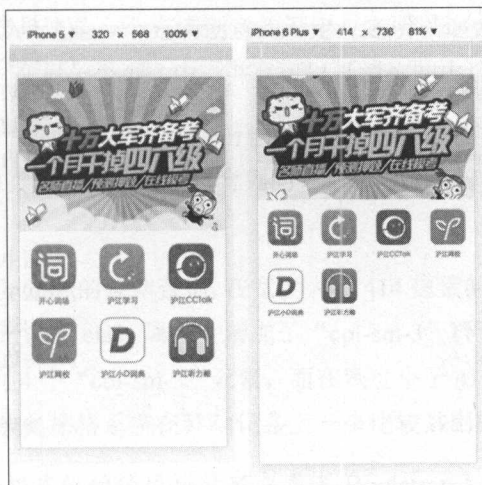


图 7.11 栅格布局在不同分辨率终端下效果

7.3.1 实现栅格布局

首先是 HTML 部分，代码如下：

```
01 <header class="banner"></header>
02 <div class="container"> /* 容器元素 */
03     <div class="row"> /* 行元素 */
04         <div class="col-xs-4 col-sm-3"> /* 列元素 */
05             <a href="#"></a> /* 图标内容 */
06             <span>开心词场</span> /* 文字内容 */
07         </div>
08         <!-- 省略代码，循环栅格列 -->
12 </div>
```

```
13 </div>
```

使用 Sass 生成不同大小的栅格, 代码如下:

```
01 @mixin clearfix() {                                /* 定义清除浮动混合宏 */
02     content: "";
03     display: table;
04     clear: both;
05 }
06 @mixin col($i) {                                    /* 定义栅格列混合宏 */
07     float: left;                                    /* 浮动 */
08     width: percentage($i / 12);                    /* 列宽 */
09     position: relative;                             /* 相对定位 */
10     min-height: 1px;                                /* 最小高度 */
11     padding-right: 15px;                             /* 内边距 */
12     padding-left: 15px;                             /* 内边距 */
13     box-sizing: border-box;                         /* 将内边距纳入宽度计算 */
14 }
15 .container {
16     margin-left: auto;                                /* 居中设定 */
17     margin-right: auto;                              /* 居中设定 */
18     padding-right: 15px;                             /* 内边距 */
19     padding-left: 15px;                              /* 内边距 */
20     &::after {
21         @include clearfix;                            /* 清除浮动 */
22     }
23 }
24 .row {
25     margin-right: -15px;                             /* 负值外边距 */
26     margin-left: -15px;                              /* 负值外边距 */
27     &::after {
28         @include clearfix;                            /* 清除浮动 */
29     }
30 }
31 @for $i from 1 through 12 {                          /* 循环生成 */
32     .col-xs-#{$i} {                                  /* 不同比例的列 */
33         @include col($i);                            /* 生成栅格列代码 */
34     }
35 }
36 @media (min-width: 414px) {                          /* 宽度大于 414px 的设备 */
37     @for $i from 1 through 12 {                      /* 循环生成 */
38         .col-sm-#{$i} {                              /* 特定设备下不同比例的列 */
```

```

39         @include col($i);                                /* 生成栅格列代码 */
40     }
41 }
42 }

```

7.3.2 栅格布局的原理

栅格系统由容器（Container）、行（Row）和列（Column）组成，行包含在容器之中，只有列可以作为行的直接子元素。比如，列设置 15 像素的内边距，从而创建了列与列之间的间隔，也称作槽（Gutter）。需要特别说明的是，由于行被设置了负 15 像素的外边距从而抵消了容器的内边距，使整个行可以充满容器，这是一个充满创意的设定，之所以这么做，是因为当采用多层嵌套的栅格布局时，行可以包含在另一个列之中，并且可以抵消列的内边距。

实例 Sass 代码中的行被平均划分为 12 个等份也就是 12 栅格列，并且预置了前缀为“.col-xs-”的 12 个能包含不同数量列的类，当行中元素包含列的个数超过 12 个时，多余列所在的元素将会另起一行排列。

Sass 代码第 36~42 行结合 CSS 的媒体查询，在宽度大于 414 像素的设备上预置了“.col-sm-”这 12 个类，并且在 HTML 元素的 class 属性上增加了“col-sm-3”样式类，因此在宽度大于 414 像素的设备上，每行可以排列 4 个“col-sm-3”元素，而在宽度小于 414px 的设备上，每行可以排列 3 个“col-xs-4”元素，这也是栅格系统在移动设备上一个比较常用的使用场景。

注意：关于栅格系统一些其他的使用场景可以参考 Bootstrap 的官方文档。在偏向移动端的应用中也可以结合 rem 来使用栅格，具体使用方法可以参考 GitHub 的 lib-flexible 项目，地址为 <https://github.com/amfe/lib-flexible>。

7.4 Flex 多栏布局实战

本实例采用 Flex 实现多栏布局，在不同分辨率下，显示不同的效果。在屏幕宽度大于 1000 像素的时候，显示为三栏，效果如图 7.12 所示。

当屏幕宽度小于 1000 像素，大于 768 像素时，隐藏右侧边栏，效果如图 7.13 所示。

当屏幕宽度小于 768 像素时，纵向排列三栏，效果如图 7.14 所示。

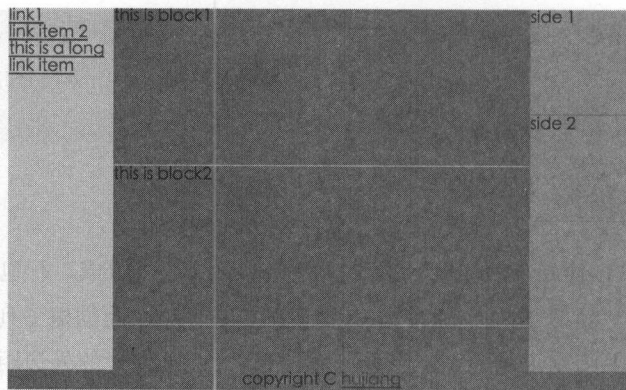


图 7.12 显示三栏布局

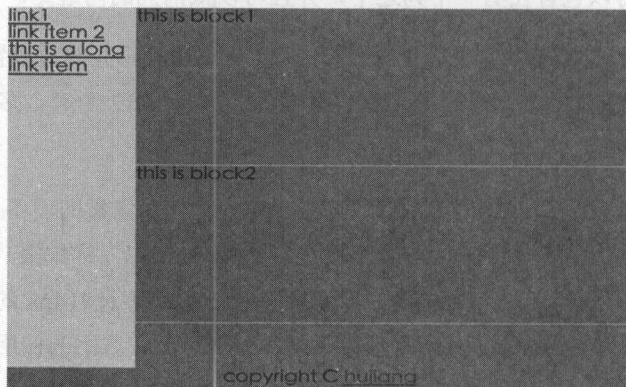


图 7.13 隐藏右侧边栏

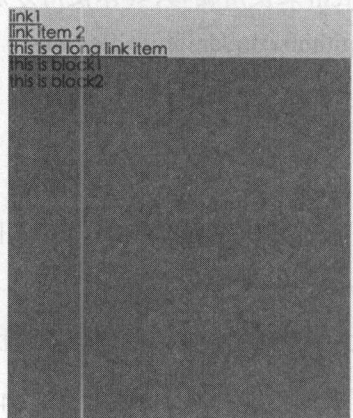


图 7.14 纵向排列三栏

首先,要实现图 7.12 的效果,HTML 代码如下:

```
<body>
  <div class="container">                                <!-- 页面主体 -->
    <nav>                                                <!-- 导航菜单 -->
      <div><a href="#">link1</a></div>
      ...
    </nav>
    <main>                                                <!-- 页面内容主体 -->
      <div class="block1">this is block1</div>
      <div class="block2">this is block2</div>
    </main>
    <aside>                                              <!-- 侧边栏 -->
      <div class="side1">side 1</div>
      <div class="side2">side 2</div>
    </aside>
  </div>
  <footer>                                              <!-- 页面底部 footer -->
    copyright C <a href="//www.hujiang.com">hujiang</a>
  </footer>
</body>
```

在图 7.12 中,首先实现底部 Footer 元素的效果,采用 Flex 布局,代码如下:

```
body{
  display: flex;          /* 整体设置为 flex 布局 */
  flex-flow: column;      /* 设置子项的排列方向为纵向排列 */
  min-height: 100vh;      /* 设置最小高度 */
}
.container{
  flex: 1;                /* 上部主容器自适应高度 */
  display: flex;          /* 主容器内部也采用 flex 布局,实现导航、内容主体和侧边栏的布局 */
}
.footer{
  height: 40px;           /* 固定底部高度 */
}
```

在上述代码中设置 Body 元素为 Flex 布局,并且设置 Flex 子项目的排列顺序为纵向排列,通过固定底部高度实现 Footer 元素贴近页面底部的效果。

然后设置导航菜单、内容主体和右侧边栏的布局,这三栏同样采用 Flex 布局,代码如下:

```
nav, aside{
```

```

width:200px;           /* 导航和侧边栏固定宽度 */
min-width: 200px;
max-width: 200px;
height: auto;          /* 高度自适应 */
}
main{
  flex: 1;              /* 内容自适应宽度 */
}

```

实现图 7.13 需采用媒体查询功能, 代码如下:

```

media (max-width:1000px){  /* 在屏幕宽度不大于 1000px 时 */
  aside{
    display: none;        /* 隐藏右侧边栏 */
  }
}

```

实现图 7.14 的效果, 需要设置上部主体容器的 Flex 布局项目的排列顺序, 代码如下:

```

@media(max-width:768px){  /* 在屏幕宽度不大于 768px 时 */
  container{
    flex-direction: column; /* 设置排列方式为纵向排列 */
  }
  nav, aside{              /* 覆盖前面定义的侧边栏和导航栏的宽度, 设置为屏幕宽度 */
    width: 100%;
    min-width: 100%;
    max-width: 100%;
  }
  aside{                   /* 覆盖在前一个@media 中隐藏的侧边栏设置 */
    display: block;
  }
  main{                   /* 纵向排列之后, 手动设置主体内容的高度 */
    min-width: 90vh;
    height: 90vh;
  }
}

```

本实例采用了 Flex 布局和媒体查询相结合的方式, 实现了在不同屏幕分辨率下的多栏布局, 从而可以实现采用一套代码, 实现在 PC、平板和手机上良好的展现效果。

7.5 实战演练: 沪江网校首页 rem 布局实践

沪江网校触屏首页 <http://mc.hujiang.com> 使用了 rem 布局方式, 开发时的设计稿 iPhone 5 的尺

寸为 320 像素乘以 568 像素，同时运行时元素尺寸会随着设备尺寸的变化按比例缩放。

注意：触屏首页是比较老的页面，设计稿的尺寸是 iPhone 5 尺寸。现在沪江触屏新页面的设计稿尺寸已经切换为 iPhone 6 的尺寸。

在页面的 HTML 里加入 Meta 标签设置浏览器视窗的宽度，代码如下：

```
// 设置视窗宽度为浏览器宽度，并且禁止用户缩放
```

```
<meta name="viewport" content="initial-scale=1.0, minimum-scale=1.0, maximum-scale=1.0, user-scalable=no">
```

注意：此处的设置是视窗宽度跟浏览器的宽度保持一致，而不是跟设备宽度保持一致。

有些网站会在设置 viewport 属性时加入“width=device-width”，使得视窗宽度等于设备宽度。在移动端，大部分情况下，浏览器宽度等于设备宽度，但在 Hybrid APP 内会有例外情况。举个例子，沪江网校 iPad 版，选课中心是原生 APP 页面，当单击具体课程时，弹出原生对话框，原生对话框里展示的是内嵌的 Web 页面。这时，浏览器宽度是对话框的宽度，并不是 iPad 的宽度，如图 7.15 所示。

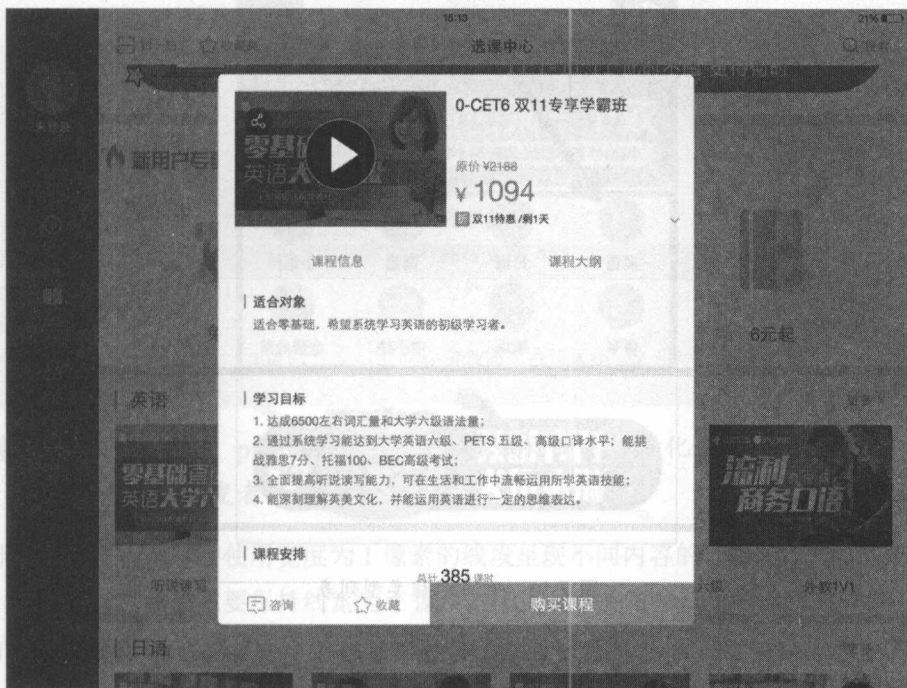


图 7.15 iPad 版选课中心页

当页面刚开始加载时, 会执行如下代码:

```
// 获取<html>元素
var documentElement = document.documentElement;
// 设置<html>元素的字体大小为视窗宽度的 1/16
documentElement.style.fontSize = documentElement.getBoundingClientRect().width / 16 + 'px';
```

以 iPhone 5 为例, 通过设置 Meta 标签, JavaScript 获取的视窗宽度为 320 像素, HTML 元素的字体大小会被设置为 20 像素。

在书写 CSS 代码时, 会将设计稿上的尺寸从单位 px 转换成 rem, 代码如下:

```
@function px2rem($x) {
  @return $x / 40 * 1rem;    // 按 640px 的设计稿, 分成 16 等份, 每等份 640 / 16 = 40px
}
```

如图 7.16 箭头所示的图标, 设计稿尺寸为 66 像素乘以 66 像素, 通过 px2rem 函数处理之后, CSS 里的尺寸会设置为 1.65rem 乘以 1.65rem。



图 7.16 首页课程类型列表

如果页面运行在 iPhone 5 上, 因为 HTML 元素的字体大小被设置为 20 像素, 图标的尺寸从 rem 换算成 px, 值为 33 像素乘以 33 像素, 完美还原设计稿。如果是在其他尺寸的设备上, 图标尺寸会等比例缩放。

图 7.16 方框所示效果 HTML 部分核心代码如下:

```

<div class="entry">                                <!-- 所有课程 -->
  <a class="inline-box">                            <!-- 课程可单击, 语义化 -->
    <i class="icon-lan"></i>                        <!-- 课程图标 -->
    <p></p>                                          <!-- 课程名称 -->
  </a>
  /* .....省略*/
</div>

```

CSS 核心代码如下:

```

.entry {                                           /* 所有课程 */
  padding-top: px2rem(8);                         /* 上内边距 */
  margin-bottom: px2rem(8);                       /* 下外边距 */
  background: #fff;                               /* 背景颜色 */
  text-align: center;                             /* 图标整体水平居中对齐 */
}
.inline-box {                                     /* 课程 */
  font-size: px2rem(10);                          /* 字体大小 */
  margin-bottom: px2rem(8);                       /* 下外边距 */
  float: left;                                    /* 使用浮动是元素排列同一行 */
  width: 25%;                                     /* 一行排列 4 个图标 */
}
.icon-lan {                                       /* 课程图标 */
  width: px2rem(33);                              /* 图标宽度 */
  height: px2rem(33);                             /* 图标高度 */
  margin-bottom: px2rem(8);                       /* 下外边距 */
  background-image: url();                        /* 背景图标 */
  background-size: 100% 100%;                    /* 背景图标占位大小 */
}

```

注意: 在实际项目中, px2rem 的使用可以在编译时由自动化工具处理, 不必像实例代码一样每次由开发者书写。

设计师喜爱在设计稿中使用宽度为 1 像素的线段呈现不同内容的分隔, 如图 7.17 所示。这种情况下, 通常设计师的原意要保持线宽为 1 像素, 所以尺寸会直接设置为 1 像素, 与设备尺寸无关。

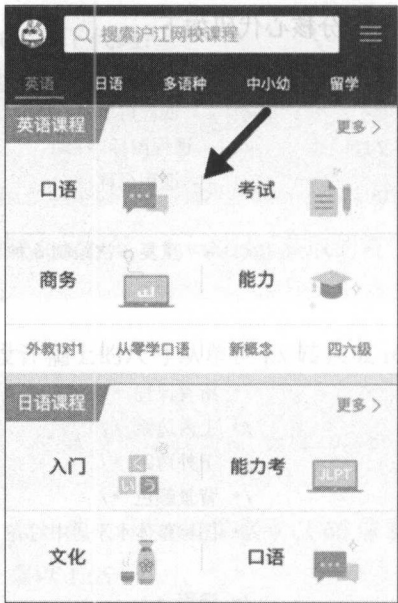


图 7.17 1 像素线

7.6 实战演练：侧边栏的滑进滑出效果

在移动开发中，由于屏幕空间有限，经常会把一些菜单或功能浮窗隐藏在屏幕一侧，当单击展开按钮时，以滑出的效果呈现内容。本节将实现一个滑进滑出效果的侧边栏实例。为了简化代码，实例中采用沪江网校首页的截屏作为主体内容，如图 7.18 所示。

单击图 7.18 左上角用户头像时，侧边栏快速滑出，然后单击右侧区域，侧边栏再快速滑入，滑出后效果如图 7.19 所示。

从图 7.19 中可以看出，侧边栏悬浮在网页主体内容之上，通过指定元素样式 `position` 属性值为 `absolute` 实现该效果。接下来要将其隐藏，一般情况下绝对定位通过指定 `left` 和 `top` 来实现，此处只需要将 `left` 属性赋值为“元素宽度乘以负 1”即可。弄清楚这些基本实现思路，再来编写代码就容易多了。

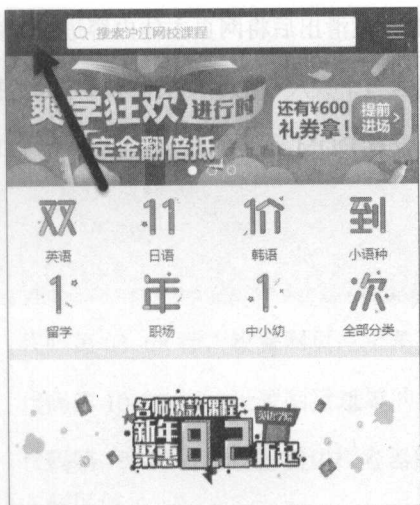


图 7.18 实例网页默认展示效果

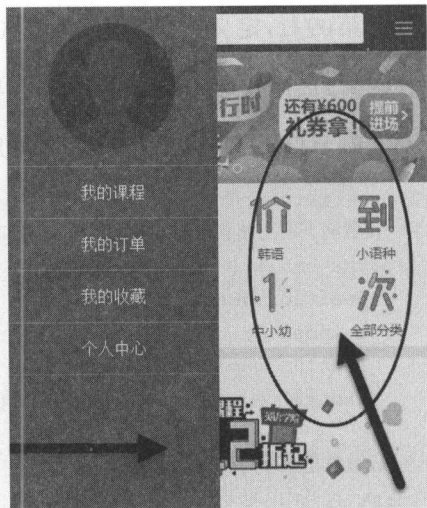


图 7.19 滑出侧边栏效果

首先定义 HTML 结构，代码如下：

```

01 <!-- 定义网页主体内容的容器 -->
02 <div class="body">
03     <!-- 定义左上角用户头像 -->
04     <a class="avatar_small"></a>
05     <!-- 网页主体内容 -->
06     
07 </div>
08 <!-- 定义网页遮罩 -->
09 <div class="mask"></div>
10 <!-- 定义侧边栏容器 -->
11 <div class="nav">
12     <ul>
13         <li><a></a></li>
14         <li><a>我的课程</a></li>
15         <li><a>我的订单</a></li>
16         <li><a>我的收藏</a></li>
17         <li><a>个人中心</a></li>
18     </ul>
19 </div>

```

HTML 结构分为主体、遮罩和侧边栏三个部分。而遮罩和侧边栏默认隐藏。

代码第 02~07 行定义网页的主体内容，该网页包含一个用户头像按钮和一张沪江网校的截图。

代码第 09 行, 定义一个透明度为 10% 的遮罩层, 用于侧边栏滑出后将网页主体内容进行遮盖。

代码第 11~19 行定义侧边栏容器和内容, 侧边栏内容包含一个大尺寸的用户头像和 4 个超链接。

定义好 HTML 结构之后, 还需要对应的 CSS 代码来设置 DOM 元素的显示, CSS 代码如下:

```

01 body { margin:0; }
02 /* 设置网页左上角头像样式 */
03 .avatar_small {
04     position:absolute;          /* 设置定位方式 */
05     z-index:2;
06     left:0; top:0;              /* 设置位置 */
07     width:1rem; height:1rem;    /* 设置大小 */
08 }
09 /* 设置滑出菜单时网页遮罩的样式 */
10 .mask {
11     display:none;               /* 默认隐藏 */
12     position:fixed;             /* 设置定位方式 */
13     z-index:10;
14     left:0; top:0;              /* 设置位置 */
15     width:100%; height:100%;    /* 设置大小 */
16     background-color:rgba(0,0,0,0.1); /* 设置背景色和透明度 */
17 }
18 /* 设置侧边栏样式 */
19 .nav {
20     position:absolute;          /* 设置定位方式 */
21     z-index:11;
22     left:-5rem; top:0;           /* 设置初始位置 */
23     width:5rem; height:100%;    /* 设置大小 */
24     background-color:#555;      /* 设置背景色 */
25 }
26 /* 设置侧边栏滑出效果 */
27 .nav {
28     transition:left linear 0.3s; /* 设置过渡函数和时间 */
29 }
30 /* 清除 UL 和 LI 的默认样式 */
31 .nav ul, .nav li {
32     list-style:none;
33     padding:0;
34     margin:0;
35 }
36 /* 设置侧边栏超链接样式 */

```

```

37 .nav a {
38     display: block;
39     padding: 1em 0;
40     border-bottom: 1px solid #888;
41     font-size: 16px;
42     color: #eee;
43     text-align: center;
44 }
45 .nav .avatar_big { width: 3rem; }

```

代码第 03~08 行，设置网页左上角头像按钮的大小和位置。

代码第 10~17 行，设置网页遮罩的样式，背景黑色且透明度为 90%，默认隐藏。

代码第 19~25 行，设置侧边栏容器的样式，宽度为网页的 50%，高度为 100%，且默认隐藏在网页左侧区域。

代码第 27~29 行，利用 CSS 3 新特性设置侧边栏滑出的效果，以线性过渡方式在 0.3 秒内将样式 left 的值改变到指定数值。

代码第 31~35 行，清除标签 UL 和 LI 的浏览器默认样式。

代码第 37~44 行，设置侧边栏容器里超链接的样式。

完成了页面的结构和样式，接下来实现滑出效果部分，JavaScript 代码如下：

```

01 // 分别获取元素：用户头像、遮罩和侧边栏
02 var btn = document.querySelector('.avatar_small'),
03     mask = document.querySelector('.mask'),
04     nav = document.querySelector('.nav');
05 btn.addEventListener('click', function () { // 给用户头像添加单击事件
06     mask.style.display = 'block'; // 展示遮罩
07     nav.style.left = '0'; // 滑出侧边栏
08 }, false);
09 mask.addEventListener('click', function () { // 给遮罩添加单击事件
10     mask.style.display = 'none'; // 隐藏遮罩
11     nav.style.left = '-5rem'; // 隐藏侧边栏
12 }, false);

```

代码第 02~04 行，分别获取用户头像、遮罩和侧边栏这三个元素，以备后续编写逻辑代码需要。

代码第 05~08 行，为用户头像添加单击事件，当用户单击左上角头像时，通过修改遮罩和侧

边栏的 CSS 将其显示。

代码第 09~12 行, 为遮罩添加单击事件, 当用户单击侧边栏之外的区域时, 再将遮罩和侧边栏隐藏起来。

注意: 实例中的遮罩看似多余, 但在实际项目中非常有必要, 一是可以保护页面元素不被误点, 二来直接将隐藏侧边栏的事件绑定在遮罩层元素上, 极大地简化了实现难度, 也降低了维护成本。

7.7 实战演练: 模拟原生的页面切换效果

本节使用 CSS 3 Transform 模拟原生应用的横向页面切换效果。由于 Web 应用中, 页面跳转会导致资源重新加载, 可能会产生长时间的白屏等待, 所以为了实现仿原生应用的页面切换效果, 需要以单页 Web 应用的形式来呈现, 也就是 SPA (Single Page web Application), 本节也将以极简的方式实现一个基本的单页路由, 如图 7.20 所示。



图 7.20 模拟原生页面切换过渡效果

7.7.1 实现页面切换过渡效果

首先是 HTML, 由于本节的重点是 CSS 的 Transform 以及单页路由的搭建, 多个页面的 HTML 部分可以存放在同一个 HTML 文件里, 也可以通过 JavaScript 动态创建, 无论怎么处理, 都不会对最终效果产生过多影响, 因此, 这里以全部页面存放在一个 HTML 文件中为例, 给出基本的 HTML 结构, 代码如下:


```

01 <div class="pages">
02     <div class="page page-home">
03         <a href="#/start">去游戏开始页</a>
04     </div>
05     <div class="page page-start"></div>
06     /* .....省略代码, 详见光盘源码 */
07 </div>

```

<!-- 实例页面一首页 -->
 <!-- 页面链接 -->
 <!-- 实例页面一游戏开始页 -->
 <!-- 更多页面 -->

CSS 部分, 使用 Transform 实现一个横向过渡效果, 代码如下:

```

01 .page {
02     height: 100%;
03     -webkit-overflow-scrolling: touch;
04 }
05 .page > div {
06     background-color: #fff;
07     height: 100vh;
08 }
09 .enter {
10     position: absolute;
11     top: 0;
12     right: 0;
13     bottom: 0;
14     left: 0;
15     z-index: 1;
16     animation: slideIn .5s forwards;
17 }
18 @keyframes slideIn {
19     from {
20         transform: translate3d(100%, 0, 0);
21         opacity: 0;
22     }
23     to {
24         transform: translate3d(0, 0, 0);
25         opacity: 1;
26     }
27 }

```

/* 页面高度 */
 /* 开启回弹 */
 /* 背景色 */
 /* 子层高度 */
 /* 使用绝对定位 */
 /* 顶边距离 */
 /* 右边距离 */
 /* 底边距离 */
 /* 左边距离 */
 /* z 轴层级 */
 /* 设置 keyframes 动画参数 */
 /* 开始状态 X 轴偏移 */
 /* 开始状态透明度 */
 /* 结束状态 X 轴到达原点 */
 /* 结束状态透明度 */

接下来, 使用 JavaScript 实现一个简单的路由功能, 代码如下:

```

01 / * 极简路由 */
02 var Router = function() {
03     this._routers = [];

```

// 构造函数
 // 路由队列


```

04     this._get = function(hash) {                                // 返回匹配路由
05         var router = this._routers.filter(function(item){      // 查询队列
06             return '#' + item.url === hash;                    // 匹配
07         });
08         return router[0] || {};                                // 返回
09     }.bind(this);                                              // 改变内部 this 指向
10 };
11 /**
12  * 推入 routers 队列
13  * @param {Object} route
14  * @return {Router}
15  */
16 Router.prototype.push = function(route) {                      // 推入队列方法
17     this._routers.push(route);                                // 推入一个路由对象
18     return this;                                              // 返回 this 实现链式调用
19 }
20 /**
21  * 路由初始化
22  * @return {Router}
23  */
24 Router.prototype.init = function() {                          // 路由初始化方法
25     this.go(this._get(location.hash));                        // 对匹配路由进行跳转
26     window.addEventListener('hashchange', function(){        // 监听 hash 变化
27         this.go(this._get(location.hash));                    // 变化时进行跳转
28     }).bind(this, false);                                    // 改变内部 this 指向
29     return this;                                              // 返回 this 实现链式调用
30 }
31 /**
32  * 路由跳转
33  * @param {String} page
34  * @return {Router}
35  */
36 Router.prototype.go = function(page) {                        // 路由跳转方法
37     var enter_page = document.querySelector(page.selector);   // 查找进场 DOM 元素
38     if (!enter_page) return false;                            // 未查到结果终止
39     enter_page.classList.add('enter');                        // 加入进场动画
40     if (page.hasOwnProperty('handle')) {                      // 是否存在 handle 成员
41         page.handle.call(this);                               // 调用 handle 方法
42     }
43 }

```

使用这个简单的路由，装载 2 个需要切换的页面，代码如下：

```

01 var home_page = {                                // 首页
02     url: '/',                                     // 对应路径
03     selector: '.page-home',                       // 对应层名
04     handle: function() {                          // 进场时调用的方法
05         console.log('Home page enter.');
```

```

06     }
07 };
08 var start_page = {                                // 游戏开始页
09     url: '/start',                                 // 对应路径
10     selector: '.page-start',                       // 对应层名
11     handle: function() {                          // 进场时调用的方法
12         console.log('Start page enter.');
```

```

13     }
14 }
15 var r = new Router();                             // 创建路由实例
16 r.push(home_page).push(start_page).init();         // 装载

```

7.7.2 模拟切换原理解析

由于页面切换过程中，进场页面通过覆盖原场景页面上层进入，因此 CSS 代码部分，进场动画“.enter”样式类使用 position 赋值 absolute 进行绝对定位，并设置了名为“slideIn”的转场动画，动画以 X 轴偏移的透明效果开始，逐渐过渡到不透明和回归 X 轴原点，并完全遮挡掉原来在场中的下层页面，从而实现了模拟原生页面的切换效果。

注意：使用 keyframes 帧动画时，在某些特定场合，仍然需要添加相应的浏览器前缀。

路由代码部分解析如下。

首先，在 Router 构造函数的内部，定义 get 方法返回路由队列里与 hash 匹配的路由对象。

然后，在 Router 函数的原型对象上添加 push 方法，负责将页面对象装载进路由队列中保存，添加 go 方法对进场页面加入“.enter”类实现进场效果，添加 init 开启一个 hashchange 事件监听，并在监听到 hash 变化时调用路由的跳转方法。由于 location 的 hash 变化，并不会导致页面的重新加载，因此当调用 go 方法对进场页加入“.enter”样式类时，可以顺利完成预置的动画效果及页面的切换。

另外，go 方法也会调用进场页面的 handle 方法，在 handle 方法中，可以使用 JavaScript 完成一些其他的工作，比如使用 AJAX 请求异步数据及进行 DOM 操作等。

最后,只要在 HTML 页面中加入一个带有 hash 的链接,如同本例中 HTML 代码部分的“#/start”,当单击这个链接时,就可以完成页面切换了。

另外,也可以使用 History API 提供的 `pushState` 和 `replaceState` 来实现路由功能,但使用 hash 相比使用 History API 在兼容性方面表现得更为突出,这一点不仅仅是在原始的 Internet Explorer 桌面浏览器上,在一些流行的移动端 APP 中,使用 History API 也可能会导致一些意想不到的问题。

注意: 本例只是讲解模拟原生页面切换效果,实例中所使用的路由相比实际应用中使用的路由还有很多需要完善的地方,比如匹配 hash 时,并没有使用正则表达式匹配类似 Node.js Web 应用框架 Express 的 URL 的路径。读者可以参考一些成熟框架使用的路由,比如 AngularJS 或 Backbone 的路由,来实现一套自己的路由功能。

7.8 提高 Web 动画的性能实战

在网页动画出现初期,许多开发者选择使用 jQuery 这些类似的库为页面添加动画效果,但是这些类库为了更好的兼容性或一些其他的目的,并没有对动画进行优化,导致动画往往十分卡顿,使得用户对网页动画产生不好的印象。随着新型浏览器的普及和开发者对优秀效果体验的重视,Web 动画无论在效果还是流畅度上都得到了大幅提升,本节将提供读者一些提高 Web 动画性能的实现方法。

7.8.1 使用 CSS 3 动画

为什么 CSS 3 动画相对于早期的 JavaScript 动画有着很大的性能提升,这要从 CSS 的图层概念说起,浏览器中会将页面切分为多个图层,每个图层有一个或多个节点。在渲染 DOM 的时候,浏览器实际的工作是由上到下顺序执行的,如下所示:

- 获取 DOM 后分割为多个图层。
- 对每个图层的节点计算样式结果。
- 为每个节点生成图形和位置。
- 为每个节点绘制填充到图层位图中。
- 图层作为纹理上传至 GPU。
- 符合多个图层到页面上生成最终页面。

另一个重要的特性是每个图层中的元素尺寸变化都会影响到别的元素。比如说,夹在两个兄弟元素之间的一个元素宽度缩小,那么兄弟元素的绝对定位就会动态改变,从而保持在动画元素

的旁边。很多 CSS 属性，例如：top、right、bottom、left、margin、padding、border 的厚度以及 width 和 height，一旦改变就会造成临近元素尺寸或位置的修改。而这种修改会导致整个图层的新布局，如果频繁操作会产生画面卡顿。

CSS 3 动画则会规避这个问题，开发者需要设置 Transform 属性，在一些先进的浏览器中，该属性会触发一个新的图层，甚至会启动设备的硬件加速（可以通过设置“translateZ(0)”或者“translate3d(0,0,0)”触发），这样性能消耗的主要点仅仅集中在了图层的组合上。

样式 Opacity 也是一个会触发 GPU 加速的 CSS 属性，所以一些例如改变 Color 属性的动画，可以使用这一属性进行代替。

7.8.2 使用高性能的 JavaScript 动画

虽然 CSS 3 动画有着不错的性能，但在开发过程中有时不得不选择 JavaScript 实现动画效果。首先 CSS 3 动画缺乏强大的控制能力，只能完成一些预设好的效果，而 JavaScript 可以在其动画过程中对其进行控制，有一些复杂的动画只能通过 JavaScript 来实现，然而 JavaScript 动画很多时候会导致页面卡顿。由于 JavaScript 动画发生在浏览器的主线程中，该线程通常十分繁忙，如果设置不当很有可能发生丢帧的情况，下面提出几个优化点能够规避一些问题。

1. 去除布局颠簸

开发者为一个元素进行持续地获取和设置操作时，在每次设置过程中，浏览器会计算这次更改所产生的后续影响（每个变化都会影响周围的元素），如果这个情况大量发生或者发生在循环中，就会导致 UI 性能降低，该情况被称为布局颠簸。

这类问题可以在开发的代码层面进行优化，开发者需要将 DOM 设置和获取的操作分别集合在一起，比如说如下代码：

```
var curTop = $("element").css("top");           // 获取元素的 top
$("element").style.top = curTop + 1;             // 设置新 top
var curLeft = $("element").css("left");           // 获取元素的 left
$("element").style.left = curLeft + 1;            // 设置新 left
```

这样的代码因为分离了获取和设置，使得浏览器对于同一时间的一系列操作可以优化为一个单一的操作，而这里的代码因为分离获取和设置并不能利用这一特性容易导致性能问题，可以优化为如下代码：

```
var curTop = $("element").css("top");           // 获取元素的 top
var curLeft = $("element").css("left");         // 获取元素的 left
$("element").style.top = curTop + 1;            // 设置新的 top
```



```
$("#element").style.left = curLeft + 1; // 设置新的 left
```

另外在一些循环调用中的 `setTimeout` 或者 `setInterval` 进行动画调用时,不合理的时间设置也会导致页面卡顿,这时候推荐使用 `requestAnimationFrame`,这样动画的绘制交由浏览器绘制请求,避免刷新间隔之间插入绘制请求导致丢帧出现卡顿感,调用方法代码如下:

```
var handle = setTimeout(renderLoop, period); // setTimeout 实现
var handle = requestAnimationFrame(renderLoop); // requestAnimationFrame 实现
```

2. 使用节流函数

另一个重要的优化点是使用节流函数,首先要明白为什么需要节流函数。有时候开发者会需要注册一些回调函数在浏览器的 `scroll` 或 `resize` 事件上,但是 `scroll` 或 `resize` 这类的事件触发非常频繁,而实际用户并不需要感知这么多的事件产生,造成大量函数触发,而如果这些函数与动画有关,会给浏览器造成巨大压力,节流函数可以很好地解决这个问题。

节流函数的原理是设置一个阈值,在一定时间内的触发但并不真实调用函数,从而做到性能的优化。实现的方式主要有反跳(`debounce`)和节流(`throttle`)。反跳是调用动作 n 毫秒内,才会执行该动作,若在 n 毫秒内又调用此动作则将重新计算时间。节流则是先设置一个执行周期,当调用的时刻大于等于执行周期时则执行并进入下一个周期,这里提供一个反跳的实例,代码如下:

```
function debounce(idle, func) { // 接受两个参数,间隔时间和实际调用函数
  var last; // 保存异步调用实际函数,通过闭包赋值不被销毁
  return function() {
    var ctx = this, args = arguments; // 存放函数的 this 和变量给下面的函数调用
    clearTimeout(last); // 如果该函数被调用,则清除上一个异步调用实际函数
    last = setTimeout(function() { // 重设异步调用实际函数
      func.apply(ctx, args) // 让实际函数在间隔设置的时间后调用
    }, idle)
  }
}
```

最后如果要对 JavaScript 动画进行更方便地优化,可以选择使用一些优秀的第三方动画库,比如 `Velocity.js`,这类 JavaScript 库的调用与开发者熟悉的 `jQuery` 等十分接近,而从库的层面进行的大量优化,对于开发者来说以较为轻松的方式获得了更优秀的性能。

7.9 实战演练: 课程分类列表实战

本节将利用之前介绍的 HTML 和 CSS 知识,实现一个完整的实例“沪江英语课程列表”。其

功能为,按照一定的排列顺序显示英语课程列表,效果如图 7.21 所示。



图 7.21 课程列表在不同分辨率手机下显示效果

7.9.1 实现主页结构

从图 7.21 可以看出,HTML 页面分为顶部标题、正文内容和底部页脚三块区域。本节逐一讲解。页面线框图如图 7.22 所示。

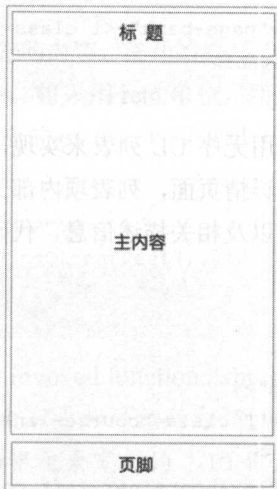


图 7.22 页面线框图

根据线框图,创建页面结构,代码如下:

```
01 <!DOCTYPE html>
02 <html>
```

```

03 <head>
04 <!-- 声明文档字符编码 -->
05 <meta charset="utf-8">
06 <!-- 移动设备 viewport -->
07 <meta name="viewport" content="width=device-width,initial-scale=1,user-scalable=no"/>
08 <title>沪江网校-英语课程</title>
09 <link rel="stylesheet" type="text/css" href="stylesheets/layout.css">
10 <head>
11 <body>
12     <div class="wrapper">
13         <header class="page-header"></header>           <!-- 标题栏 -->
14         <article class="page-body"></article>           <!-- 主内容 -->
15         <footer class="page-footer"></footer>           <!-- 页脚 -->
16     </div>
17 </body>
18 </html>

```

标题栏使用具有 HTML 5 语义的 Header 标签,在其内部包含了两个元素——回退按钮和页面标题,代码如下:

```

<header class="page-header">
    <a href="#" name="back" class="page-back"><i class="icon-back"></i></a>
    <h1 class="page-title">英语课程</h1>
</header>

```

主内容为一个课程列表,可以使用无序 UL 列表来实现,列表项 LI 实现为一个 100% 的块级元素,为了让整个列表项可以链接到详情页面,列表项内部又包含了一个 A 标签。A 标签内部分成左右分栏布局,分别展示课程图片以及相关描述信息,代码如下:

```

01 <article class="page-body">
02     <ul class="course-list">
03         <!-- 课程列表循环开始 -->
04         <li class="course-item">
05             <a href="#" title="" class="course-link flex-container">
06                 <!-- 图片区域 -->
07                 <span class="flex-container-left course-pic-container ">
08                     <!-- 图片 -->
09                     
10                 </span>
11                 <!-- 描述区域 -->
12                 <span class="flex-container-right course-desc-container">
13                     <!-- 标题 -->

```

```

14         <h2 class="course-title">0-CET6 考试保障班</h2>
15         <!-- 价格 -->
16         <h3 class="course-price">¥2449.0<del>2499.0</del></h3>
17         <!-- 详情 -->
18         <p class="course-desc">414 课时 有效期至 2018.06.29</p>
19     </span>
20 </a>
21 </li>
22 /* .....省略代码, 详见光盘源码 */
23 <!-- 课程列表循环结束 -->
24 </ul>
25 </article>

```

页脚使用 Footer 标签, 通常会放入一些版权或备案信息, 代码如下:

```

<footer class="page-footer">
    <p>Copyright 2016 © 沪江网</p>
    <p>沪江教育科技(上海)股份有限公司</p>
</footer>

```

7.9.2 响应式 CSS 实现 (Compass)

接下来开始本节的重点 CSS 部分的内容, 在开始编写 CSS 之前, 需要充分理解先前章节手机端“响应式”的实现方法。此实例中, 将采用 rem 单位, 所以需要在 HTML 根节点上加入一个基准的 font-size 值来表示 1rem 单位对应的 px 值, 可以通过 JavaScript 来实现, 代码如下:

```

;(function () {
    var docEl = document.documentElement;           // 文档根节点
    var docWidth = docEl.getBoundingClientRect().width; // 根节点宽度
    docEl.style.fontSize = docWidth / 10 + 'px';     // 设置基准字体大小
})();

```

提示: 这是一个“immediately invoked function expression (IIFE)”, 也就是“立即执行函数”, 在后面的章节会有更加详细的介绍。这段 JavaScript 脚本的功能是将页面基准的 font-size 设置为根元素宽度的 1/10 作为 1rem 的大小, 更可靠的方法是使用“lib-flexible”类库, 关于“lib-flexible”的介绍可以参阅 <https://github.com/amfe/lib-flexible>。

有了基准的 rem 以后, 就可以开始编写样式了。前面章节已经介绍, 因为浏览器的现状导致需要对一些 CSS 3 属性增加特定的前缀, 为了弥补由于对这块知识缺乏了解而导致的错误, 可以使用基于 Sass 的 Compass 框架来编写样式, Compass 提供了大量的混合宏 (Mixins) 用来处理类

似添加浏览器前缀这样的工作。

提示: 一种更优雅的处理浏览器前缀的方式是使用 CSS 的后处理程序, 比如 PostCSS 最流行的 Autoprefixer 插件, 本节暂不介绍相关内容, 读者感兴趣可以前往 GitHub 上学习, 地址为 <https://github.com/postcss/autoprefixer>。

关于 Compass 的安装, 这里不再赘述。在使用 Compass 前, 首先需要创建 Compass 的配置文件 config.rb, Compass 会根据 config.rb 文件的配置执行编译并将编译后的 CSS 文件保存到对应目录下, config.rb 代码如下:

```
css_dir = "stylesheets"      # 编译后的 CSS 文件存放目录
sass_dir = "sass"           # Sass 文件存放目录
images_dir = "images"       # 图片文件存放目录
javascripts_dir = "scripts"  # JavaScript 文件存放目录
output_style = :compressed  # 输出样式
```

也可以通命令行工具执行 Compass 的初始化命令创建 config.rb 文件, 命令如下:

```
compass init
```

接下来开始编写 Sass 代码, 可以将一些全局的变量、方法或可复用的模块单独进行提取, 便于其他模块复用。

(1) 创建一个名为 “_base.scss” 的文件, “_base.scss” 文件中会加入一些全局的变量设置、字体引用, 以及将 px 单位转化为 rem 单位的函数。由于之前已经通过 JavaScript 将页面的基准 font-size 值设置为页面根元素的 1/10 宽度, 假设设计师提供的设计稿的是根据 iPhone5 的尺寸以 640 像素为宽度, 那么在 CSS 中的 1rem 对应设计稿中的像素值应该也是设计稿宽度的 1/10, 即 64 像素。通过 pxTorem 函数, 就可以把设计稿中的任意 px 值转换成 rem 值, 代码如下:

```
$base-font-size: 64px;          // 基准字体大小
@function pxTorem($px){         // px 转 rem 函数, 参数为 px 值
  @return $px / $base-font-size * 1rem; // 返回参数对应 rem 值
}
```

提示: 设计稿中 64 像素通过 pxTorem 转换成 1rem 并写在 CSS 属性中, 而 CSS 中的 1rem, 由于基准的 font-size 的作用, 在 iPhone5 (320 像素 × 568 像素) 中, 被渲染为 32 像素, 而在 iPhone6(375 像素 × 667 像素) 中, 被渲染成 37.5 像素, 由于只是做了同比例缩放, 并始终保持了与设计稿相同的比例, 因此也就做到了响应式适配。

(2) 开始编写课程列表的样式, 新建一个 layout.scss 文件, 在 layout.scss 文件的开始, 首先使用 “@import” 命令导入刚才编写的 “_base.scss” 以及 Compass 预置的混合宏。

“@import 命令”可以分成多行，也可以写在一行，代码如下：

```
@import "base", "compass/reset", "compass/CSS 3", "compass/utilities";
```

使用代码“@import compass”可以导入所有的 Compass 预置混合宏，也可以只按需求分别导入，实例使用以下 3 种 Compass 混合宏。

- reset: 样式重置混合宏。
- CSS 3: CSS 3 的浏览器前缀混合宏。
- utilities: 一些基本的混合宏（Clearfix 等）。

(3) 接下来按之前定义的 HTML 结构来分别完成顶部标题、正文内容和底部页脚三块区域的样式开发。

顶部标题部分，CSS 代码如下：

```
01 .page-header{
02     background-color: $color_black;           // 背景色
03     width: 100%;                             // 宽度
04     height: pxTorem($header_height);         // 高度
05     position: fixed;                          // 固定定位
06     z-index: 10;                             // 堆叠顺序
07 }
08 .page-back{
09     position: absolute;                       // 绝对定位
10     top: 0;                                  // 顶部距离
11     left: 0;                                 // 左侧距离
12     width: pxTorem($header_height);          // 宽度
13     height: pxTorem($header_height);         // 高度
14     text-align: center;                      // 水平居中
15 }
16 .icon-back:before{
17     color: $color_white;                     // 字体颜色
18     font-size: pxTorem($header_height / 2); // 字体大小
19     content: '\e808';                        // 图标编码
20 }
21 .page-title{
22     text-align: center;                      // 水平居中
23     color: $color_white;                     // 字体颜色
24     font-size: pxTorem($header_height / 2.5); // 字体大小
25     line-height: pxTorem($header_height);    // 行高
26 }
```

提示: 这里也可以使用 Compass 的混合宏让代码更加简洁, 读者可以自行参考 Compass 相关的开发文档, 地址为 <http://compass-style.org>。

主内容的课程列表是一个左右布局的两栏结构, 实际项目中常用的布局方法有 float、inline-block 以及 Flex 等, 本节源代码中涵盖了这些布局方式的实现, 这里仅以 Flex 布局给出范例, 代码如下:

```
01 .flex-container{
02     @include display-flex;           // display-flex 混合宏
03     @include flex-direction(row);    // flex-direction 混合宏
04 }
05 .flex-container-left{
06     @include flex(4);               // flex 混合宏
07 }
08 .flex-container-right{
09     @include flex(6);               // flex 混合宏
10     text-indent: 15px;             // 文字缩进
11 }
```

Compass 的混合宏, display-flex、flex-direction 以及 flex, 编译后, CSS 代码如下:

```
01 .flex-container {
02     display: -webkit-flex;           // 带 webkit 前缀多栏伸缩布局
03     display: flex;                  // 标准多栏伸缩布局
04     -webkit-flex-direction: row;     // 带 webkit 前缀的横向伸缩
05     flex-direction: row;            // 标准横向伸缩
06 }
07 .flex-container-left {
08     -webkit-flex: 4;                // 带 webkit 前缀伸缩比率
09     flex: 4;                        // 标准伸缩比率
10 }
11 .flex-container-right {
12     -webkit-flex: 6;                // 带 webkit 前缀伸缩比率
13     flex: 6;                        // 标准伸缩比率
14     text-indent: 15px;             // 文字缩进
15 }
```

提示: Compass 提供了两种 Flex 的混合宏, 分别对应 2009 年以及 2012 年的草案, 具体可以参阅 W3C 相关文档。实际工作中, 通常会把 display-box 和 display-flex 都引入到项目中。

页脚是一段简单的文字说明, 这里不再赘述, 读者可以自行实现或者参考本节的项目源代码。

编写完成样式代码后，在命令行工具中运行 Compass 的编译命令，命令如下：

```
compass compile
```

将 Sass 代码编译成 CSS 代码，至此一个完整的课程列表就完成了，可以使用 Chrome 浏览器模拟手机端打开页面文件查看效果。

7.9.3 添加页面动态效果

完成了基本的页面效果以后，可以对页面增加一些动态效果，Compass 提供的 `animation`、`transform` 和 `transition` 混合宏用来创建诸如旋转、变形等 CSS 3 效果。以下利用这些混合宏来实现几个简单的 CSS 3 动画效果，如图 7.23 所示。

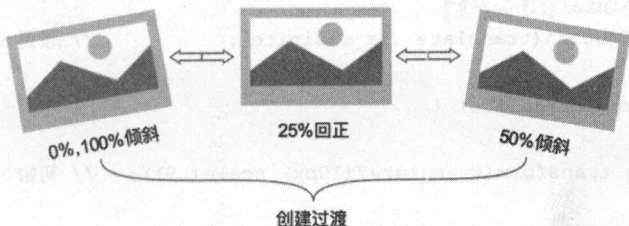


图 7.23 倾斜抖动的 CSS 3 效果示意图

倾斜抖动的 CSS 3 动态效果如图 7.23 所示，在初始（结束）和中间状态创建关键帧，设置角度偏移，代码如下：

```
01 .animation-rotate{
02     @include animation(rotate .5s infinite);           // 设置 animation 参数
03 }
04 @include keyframes(rotate) {
05     0%, 100% {
06         @include transform(rotate(-10deg) scale(.9)); // 初始角度偏移
07     }
08     50% {
09         @include transform(rotate(10deg) scale(.9));  // 中间帧角度偏移
10     }
11 }
```

完成上下跳动的 CSS 3 效果，如图 7.24 所示。

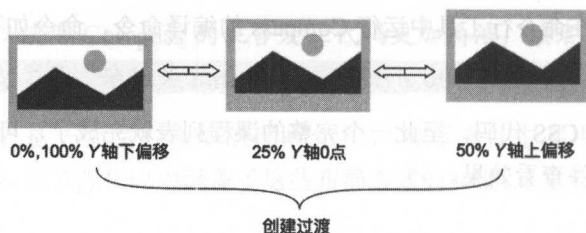


图 7.24 上下跳动的 CSS 3 效果示意图

如图 7.24 所示, 是一个上下跳动的 CSS 3 动态效果, 同样在初始 (结束) 和中间状态创建关键帧, 设置 Y 轴的偏移, 代码如下:

```

01 .animation-translate{
02     @include animation(translate .5s infinite);           // 设置 animation 参数
03 }
04 @include keyframes(translate) {
05     0%, 100% {
06         @include transform(translateY(10px) scale(.9)); // 初始 Y 轴偏移
07     }
08     50% {
09         @include transform(translateY(-10px) scale(.9)); // 中间帧 Y 轴偏移
10     }
11 }

```

提示: Animate.css 内置了很多典型的 CSS 3 动画, 读者可以使用 Compass 提供的混合宏参考 Animate.css 来实现这部分的内容, 也可以将 Animate.css 导入到项目中直接使用。Animate.css 库的使用可参考地址 <https://daneden.github.io/animate.css/>。

7.10 本章小结

移动网页布局不同于传统的 PC 端布局, 在设计和实现上需要着重考虑不同终端尺寸设备的还原, 在技术选择上越来越趋向于使用 CSS 3 新特性。CSS 3 的到来为移动端的 Web 前端技术注入了强劲的生命力, 从布局、动画、性能诸多方面都提供了全新的解决方案。本章实战内容主要介绍了实际场景中 9 种常见的使用, 希望能给读者在开发中带来帮助。

8

第 8 章

前端工程化实战

“软件工程”是一门研究如何系统化、规范化、数量化地开发和维护软件的学科，包括两方面的内容：软件开发技术和软件项目管理。其中，软件开发技术包括开发方法、软件工具和开发环境。

软件工程的目标是：在给定成本、进度的前提下，开发出具有适用性、有效性、可修改性、可靠性、可理解性、可维护性、可重用性、可移植性、可追踪性、可互操作性和满足用户需求的软件产品。以此为目标将有助于提高软件产品的质量和开发效率，减少维护的成本。

提示：软件工程定义和目标参考“软体工程（Software Engineering;SE）”，地址为 <http://irw.ncut.edu.tw/peterju/se.html>。

8.1 前端工程化

在软件开发领域，工程化必不可少。没有工程化的体系，效率、质量、合作和维护等便无从

谈起。随着互联网近几年的发展,如今的项目越来越庞大,用户交互越来越复杂,前端开发所涉及的内容也越来越多,加之移动互联网以雷霆万钧之势进入大众的生活,前端工程化变得前所未有的重要。

8.1.1 前端工程化的必要性

在很多开发者眼中,后端开发进行工程化和体系化的尝试理所当然,而前端只是一些 UI 效果和用户交互的简单实现,既没有复杂的业务逻辑,也不涉及数据库操作,因此并不需要走工程化道路。然而事实并非如此。

对于一个 Web 网页来说,除了漂亮的界面、炫酷的特效和复杂的交互之外,还有非常重要的一点:页面的渲染速度。为了将页面快速地呈现出来,通常做如下优化。

- 减少请求数:合并多个 CSS、JavaScript 文件,合并多个背景图片,使用 CSS3 图标、Icon Font 等。
- 减少请求量:压缩 CSS、JavaScript 文件内容,降低图片质量,延迟加载图片,减少 Cookie 携带,开启 GZip,使用浏览器缓存技术等。

在项目的开发过程中,这些优化工作往往会占用开发者相当多的时间,然而从工程化的角度来看,诸如此类的,大量的重复性工作是不应该出现在开发环节中的。

不仅如此,项目中还会有另外一些使开发者苦不堪言的问题,如下:

- 在新业务的驱使下,代码复杂度越来越高,维护成本直线上升。
- 随着时间的推移,代码量越来越大,网页性能慢慢下降。
- 不同开发者随意引入第三方库和组件,造成代码之间彼此冲突和大量冗余。
- CSS 命名冲突,或因写法过于全局化,造成样式莫名新增或丢失。

以上种种问题的解决方案多种多样且各不相同,其中,工程化就是解决上述问题的有效方法之一。工程化不分项目大小、人员多寡,均可以通过以下方法达成。

1. 规范代码

代码的统一规范可以避免低级错误,降低维护成本,更是团队合作的基础。在开发过程中既可以自定义代码规范,也可以遵循一些已有的开源规范,如 Airbnb Style Guide 项目(地址为 <https://github.com/airbnb/javascript>),还可以利用 JSLint 在提交代码之前对代码进行自动的 Code Lint,从而找出潜在的 BUG 和不良代码。

2. 进行 JavaScript 预处理

为了弥补 JavaScript 设计上的不足,出现了很多试图代替 JavaScript 的语言,如 CoffeeScript、LiveScript、TypeScript 等。这些替代品相比原生的 JavaScript 在语法上更有优势,编写起来更为顺畅,代码可读性也更高。然而最终它们都需要被编译为原生 JavaScript。

模块化是前端开发一直以来的追求,目前比较流行的模式有 AMD、UMD 和 CommonJS。其中 CommonJS 的风格需要编译,最终会将多个模块编译为一个以 UMD 方式定义的模块。

除了以上这些,JavaScript 在 2015 年 6 月正式发布了新的语言版本 ECMAScript 6.0,简称 ES6。ECMAScript 6.0 是 JavaScript 语言的下一代标准,目标是使 JavaScript 语言可以编写复杂的大型应用程序,成为企业级开发语言。然而目前多数浏览器对 ECMAScript 6.0 的支持并不友好,为了在浏览器上运行通过 ECMAScript 6.0 标准编写的 JavaScript 代码,需要将这些代码编译成上一个 JavaScript 版本 ECMAScript 5.0。

3. 进行 CSS 预处理

类似 JavaScript 的情况,CSS 也出现了一些对开发者而言更为友善的替代语言,如 Less 和 Sass。这些语言最终也会被编译为浏览器能够解析的原生 CSS 代码。除此之外,为兼容各种浏览器在 CSS 里写的各种 Hack 代码,也可以通过工程插件进行自动化处理。

4. 自动编译

在项目开发过程中,需要刷新浏览器来查看代码修改后的效果,有些还需要增加手动编译过程。工程化之后可以使用代码监控工具实时地自动刷新浏览器,从而大大提高开发效率。

5. 缩减文件体积

无论是前端还是后端,高性能是亘古不变的追求,这点在移动端尤为重要。Web 应用的性能优化主要通过以下几点来达成。

- JavaScript、CSS 代码压缩。
- JavaScript、CSS 文件合并。
- 图片有损压缩。
- 小图片合并(精灵图、雪碧图)。

最后,前端工程化能做的绝不仅限于此,还包括代码共享、自动部署、 workflow 管理等。读者可以根据自己的需求,开发特定的工程组件,用以完善整个项目的构建。

8.1.2 前端工程化的发展史

最后, 前端工程化能做的绝不仅限于此, 还包括代码共享、自动部署、工作流管理等。读者可以根据自己的需求, 开发特定的工程组件, 用以完善整个项目的构建。

1. 石器时代

最早期的 Web 界面非常简单, 只需要实现最简单的内容呈现和表单提交。网站利用静态的 HTML 代码提供基本的浏览内容, 只有偶尔的表单提交需要利用 JavaScript 代码实现某些功能, 这个阶段是没有前端后端之分的。这个时期, 由于整个项目几乎全部由后端工程师开发完成, 因此前后端代码经常混杂在一起。例如, 开发者会直接在页面中通过符号“<% %>”嵌入后端代码。一个简单的使用 PHP 开发的页面代码如下:

```
<html>
<head>
  <title>PHP 页面</title>
</head>
<body>
<h1>
  <?php
    echo "hello word";
  ?>
</h1>
</body>
</html>
```

这种写法显然给长期的维护带来了很大的困难。

2. 铜器时代

这个阶段比较典型的改进是 Web 开发的组件化和异步加载的实现。

“组件化”是指不再把所有的代码都杂糅在一个文件里, 而是通过 frameset 和 iframe 标签来组织页面, 把 JavaScript 代码放在一个单独的后缀名为“.js”的文件里, 通过外部链接的形式引入到页面中去。

谈到异步加载, 就不得不提到 AJAX。这项跨时代的技术为 Web 开发打开了新局面, 帮助开发者提升网站性能, 优化用户体验。

也正是因为如此, Google 的在线文档和 Gmail 邮箱得到了空前的发展。随着工作量的增加, 一些前端基础框架也在这个阶段逐渐兴起。

3. 农业时代

上个时代出现的基础框架的确能够帮助开发者更好地组织代码，从而提高代码的可读性，但是随着网络产品功能越来越复杂，代码量越来越大，页面载入时所加载的大量代码必然大大降低页面的载入速度，从而影响用户体验。为解决这个问题，模块加载规范应运而生，主要有两种：AMD（Asynchronous Module Definition）和 CMD（Common Module Definition）。以此为基础，还出现了很多基于该规范的，动态加载 JavaScript 代码的框架。

在实现 AMD 规范的框架中，知名度较高的是 RequireJS。一段依赖 jQuery 实现的 Javascript 代码在 RequireJS 框架下写法如下：

```
<script src="require.js"></script>
<script>
  require(['jquery'], function ($) {
    $('a').css('color', 'red');
  });
</script>
```

4. 工业时代

在这个阶段，移动互联网蓬勃发展，Web 前端所实现的功能日益复杂，开发难度也越来越大。为了降低开发难度，前端 MVC、MVP、MVVM 框架如雨后春笋般诞生。业内把这些框架统称为 MV* 框架，其中的主流有 Backbone、AngularJS、React、Vue 等。MV* 框架结构示意图如图 8.1 所示。

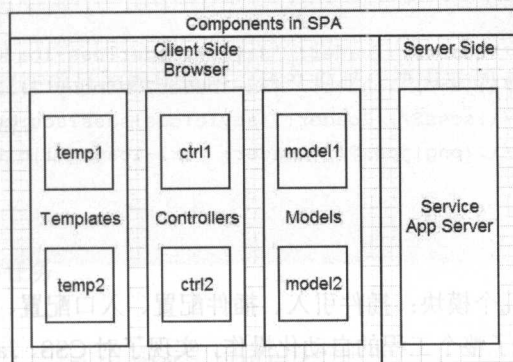


图 8.1 MV* 框架示意图

显而易见的好处是：

- 前后端分离，不再互相依赖。
- 合理的分层，让代码各司其职，降低复杂度。

同时,为了提高开发效率、加强团队合作,出现了很多以自动化为目标的构建工具,如 Grunt、Gulp 和 Webpack 等。压缩、编译、单元测试、代码检查、打包发布等重复性任务,可以使用自动化工具来降低劳动成本,简化开发者的工作。

正是这些构建工具的出现,把前端工程化推向了一个新的台阶。构建工具的核心思想是通过一个配置文件来实现所有需要自动化的功能。一个典型的 Webpack 配置文件内容如下:

```
01 var webpack = require('webpack');
02 var commonsPlugin = new webpack.optimize.CommonsChunkPlugin('common.js');
03 module.exports = {
04     //插件项
05     plugins: [commonsPlugin],
06     //页面入口文件配置
07     entry: {
08         index : './src/js/page/index.js'
09     },
10     //入口文件输出配置
11     output: {
12         path: 'dist/js/page',
13         filename: '[name].js'
14     },
15     module: {
16         //加载器配置
17         loaders: [
18             { test: /\.css$/, loader: 'style-loader!css-loader' },
19             { test: /\.js$/, loader: 'jsx-loader?harmony' },
20             { test: /\.scss$/, loader: 'style!css!sass?sourceMap' },
21             { test: /\.png|jpg$/, loader: 'url-loader?limit=8192' }
22         ]
23     }
24 };
```

基本的配置内容分为几个模块:插件引入、插件配置、入口配置、输出配置、加载器配置。短短的 20 多行代码便完成了整个工程的自动化操作,实现了对 CSS、JavaScript、Sass 和图片的预处理,最后编译出可以发布到产线的资源文件。

8.2 工程化入门 Grunt

Grunt 是一个基于 JavaScript 的强大的任务管理器,允许在终端机上完成验证 JavaScript 语法、

CSS 压缩、Sass 编译等诸如此类的任务。通过 Grunt，可以实现自动化构建、测试等，帮助团队提高开发效率、减少错误率。

8.2.1 安装和配置

Grunt 基于 Node.js 运行并通过 NPM 安装和管理，安装命令如下：

```
npm install -g grunt-cli
```

此时，Grunt 命令已经加入系统。但是，安装 grunt-cli 并不等于安装了 Grunt。Grunt CLI 的任务是调用与 Gruntfile 在同一目录中的 Grunt，这样做的好处是允许同一个系统安装多个版本的 Grunt。

注意：执行以上命令时可能需要使用 sudo 权限，Windows 环境下要作为管理员来执行该命令。

当开发者通过命令行运行 Grunt 时，它会利用 Node.js 提供的 require 方法查找本地安装的 Grunt。找到后，CLI 加载 Grunt，传递 Gruntfile 中的配置信息，并且执行 Gruntfile 中预设的任务。由此可见，Gruntfile 是 Grunt 中最为重要的文件。

Gruntfile 文件的主要作用有：

- 配置或定义任务（task）
- 加载 Grunt 插件

Gruntfile 文件需要放置在项目的根目录中。该文件是一个有效的 JavaScript 或 CoffeeScript 文件，主要由以下几部分构成：

- wrapper 函数
- 项目与任务配置
- 加载 Grunt 插件和任务
- 自定义任务

下面来看一个具体的 Gruntfile.js 文件，代码如下：

```
01 // 包装函数
02 module.exports = function (grunt) {
03     // 插件配置信息
04     grunt.initConfig({
05         pkg: grunt.file.readJSON('package.json'),
06         // uglify 插件
```



```

07     uglify: {
08         // 配置信息
09         options: {
10             beautify: true,        //是否压缩
11             mangle: false,         //是否混淆变量名
12             compress: true,        //是否使用默认选项源压缩
13         },
14         // 任务
15         app_task: {
16             files: {
17                 'build/app.min.js': ['lib/index.js', 'lib/test.js']
18             }
19         }
20     }
21 });
22 grunt.loadNpmTasks('grunt-contrib-uglify'); // 加载指定插件
23 grunt.registerTask('default', ['uglify']); // 执行指定任务
24 };

```

代码第 4~12 行是初始化 Grunt 的配置数据，这些配置数据通过一个 object 对象传递给 `grunt.initConfig` 方法。其中第 5 行代码将存储在 `pageage.json` 文件中的 JSON 元数据引入 Grunt 配置中。

代码第 7~20 行是插件“uglify”的配置信息。

代码第 22 行加载指定插件，此处需加载插件“grunt-contrib-uglify”。

代码第 23 行执行指定任务，第一个参数“default”指默认执行，第二个参数为数组，表示将要执行的任务列表。

上述代码中的另一个文件 `package.json` 是整个项目的基本信息和配置文件，也是 Node.js 项目的核心文件之一。该文件可以直接通过命令“npm init”生成，主要包含信息如下：

- 项目的基本信息，如名称、版本、作者等。
- 项目所使用的插件列表。

在接下来的章节中，将会重点介绍 Grunt 插件的使用方法。

8.2.2 Grunt 插件

如果说 Grunt 是一台电子设备，那么插件就是运行在设备上的各种软件。没有软件，设备什

么都做不了。本节将主要介绍 Grunt 插件的安装和使用，以及插件无法满足需求时的处理方法。

1. Grunt 插件的安装

如同 Grunt，插件也需要通过 Node.js 的包管理工具 NPM 安装。uglify 插件安装命令如下：

```
npm install grunt-contrib-uglify --save-dev
```

插件被安装到本地的同时，还会自动将该插件添加到 package.json 文件的 devDependencies 配置段中（遵循 tilde version range 格式）。

提示：关于 tilde version range 格式请参考文档 <https://npmjs.org/doc/misc/semver.html#Ranges>。

2. Grunt 插件的使用

在 Grunt 中使用插件需完成四个步骤：配置、加载、注册和执行。其中配置、加载和注册通过 Gruntfile 文件定义，先看该文件的插件配置信息，代码如下：

```
grunt.initConfig({
  pkg: grunt.file.readJSON('package.json'),
  uglify: {
    options: {
      banner: '/*! <%= grunt.template.today("yyyy-mm-dd") %> */\n'
    },
    build: {
      src: 'src/<%= pkg.name %>.js',
      dest: 'build/<%= pkg.name %>.min.js'
    }
  }
});
```

这段代码首先将 package.json 文件中的项目元数据导入 Grunt 配置中，这时 package.json 文件的内容会被当作一个 JavaScript 对象存储在 pkg 属性中。接下来，在 uglify 插件的配置中，引用了元数据中的 name 属性作为项目中的 JavaScript 文件名，元数据通过 “<%%>” 表示。

Grunt 加载名为 “grunt-contrib-uglify” 的插件的代码如下：

```
grunt.loadNpmTasks('grunt-contrib-uglify');
```

若加载多个插件，多次调用 loadNpmTasks 函数即可。

注册任务代码如下：

```
grunt.registerTask('default', ['uglify']);
```

当 Grunt 运行时默认执行 uglify 任务。换言之，当运行 grunt 命令时，若无指定参数，则执行通过 default 定义的任务。registerTask 函数有 2 个参数：自定义的任务名称和任务列表。任务列表是一个字符串数组，可以指定一个或多个任务。

注册完任务，通过 grunt 命令来执行任务。registerTask 函数中的 default 是 Grunt 的预设参数。以上案例中执行命令“grunt”、“grunt default”、“grunt uglify”，效果等同。

3. 开发自定义任务

很多情况下，Grunt 插件中的任务并不能完全满足项目需求，还需要借助自定义任务来完成一些操作。Grunt 自定义任务的语法如下：

```
grunt.registerTask(taskName, [description], taskFunction)
```

- taskName: 任务名称，命令行里使用 grunt <taskName>。
- description: 任务的描述。
- taskFunction: 任务的实现。

自定义任务在 Gruntfile 文件中定义，一个简单的自定义任务代码如下：

```
module.exports = function(grunt) {  
  grunt.registerTask('mytask', '自定义任务：输出参数内容', function(arg) {  
    grunt.log.writeln('任务' + this.name + "的参数是: " + arg);  
  });  
};
```

该段代码向 Grunt 注册了一个名称为“mytask”的任务，该任务将命令行输入的参数打印出来，执行任务的命令如下：

```
grunt mytask:hello
```

当然，自定义任务也不是必须在 Gruntfile 文件中定义，还可以通过外部 JavaScript 文件定义，然后通过 grunt.loadTasks 方法加载。

4. 常用的插件

Grunt 的插件非常多，有些由 Grunt 团队开发，也有一些来自社区贡献，其中常用的插件见表 8.1。

表 8.1 常用Grunt插件表格

插 件 名 称	插 件 描 述
grunt-contrib-watch	当监控的文件有改变时，运行指定的任务
grunt-contrib-clean	删除指定的文件和文件夹

续表

插件名称	插件描述
grunt-contrib-copy	复制指定的文件和文件夹
grunt-contrib-uglify	使用UglifyJS压缩JavaScript文件
grunt-contrib-jshint	使用JSHint验证JavaScript代码
grunt-contrib-concat	合并多个文件
grunt-contrib-cssmin	压缩CSS文件
grunt-karma	用于做单元测试
grunt-contrib-connect	构建一个实时预览的开发环境
grunt-contrib-less	将LESS文件编译成CSS文件
grunt-concurrent	并发运行多个Grunt任务
grunt-sass	将Sass文件编译成CSS文件
grunt-shell	支持运行shell命令
grunt-newer	仅运行源文件被修改过的任务
grunt-eslint	使用ESLint验证JavaScript代码
grunt-contrib-imagemin	对图片进行压缩
jit-grunt	按需加载Grunt插件
typedoc	为typescript项目创建API文档
grunt-text-replace	使用字符串、正则或函数替换文本

提示：可以访问官方网站 <http://gruntjs.com/plugins> 查阅插件列表。

8.2.3 实战演练：使用 Grunt 开发一个简易相册

以上章节对 Grunt 进行了基本的介绍。本节将通过实例演示如何使用 Grunt 开发项目。

该项目要实现的功能有：单击页面底部缩略图，在页面中间展示对应的高清图片。整个项目的文件结构如图 8.2 所示。

该实例会用到 4 个 Grunt 插件，如下。

- grunt-contrib-uglify：压缩 JavaScript 文件。
- grunt-contrib-cssmin：压缩 CSS 文件。
- grunt-inline：将比较小的图片文件转成 base64 格式。
- grunt-contrib-concat：将多个静态文件合并成一个。

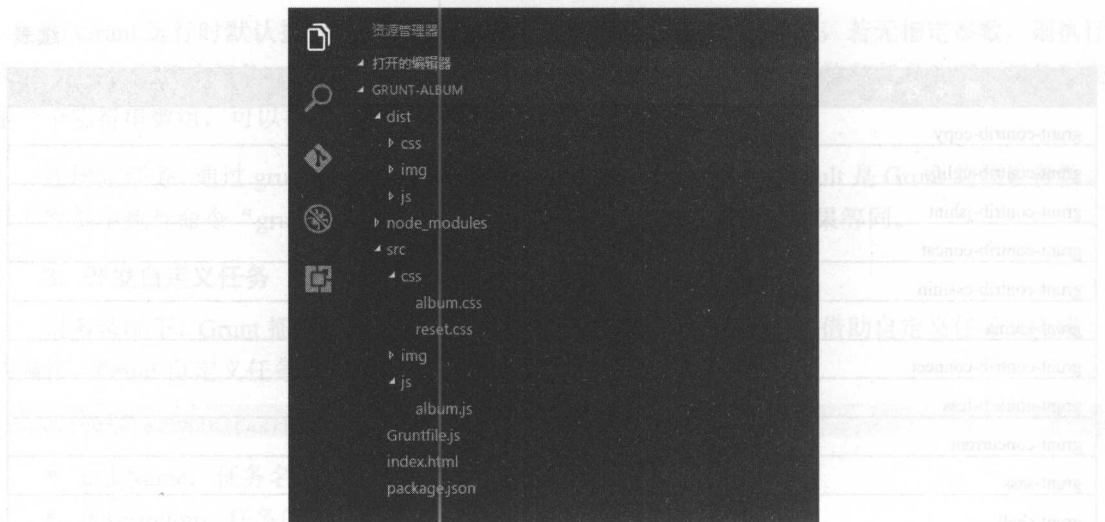


图 8.2 Grunt 项目代码结构截图

插件的安装命令如下：

```
npm install grunt-contrib-uglify --save-dev
npm install grunt-contrib-cssmin --save-dev
npm install grunt-inline --save-dev
npm install grunt-contrib-concat --save-dev
```

安装插件后，系统会自动在 `package.json` 文件中添加该插件的配置，代码如下：

```
01 {
02   "name": "grunt-album-app",
03   "version": "0.0.1",
04   "description": "",
05   "scripts": {
06     "test": "echo \"Error: no test specified\" && exit 1"
07   },
08   "author": "",
09   "license": "ISC",
10   "devDependencies": {
11     "grunt-contrib-concat": "^1.0.1",
12     "grunt-contrib-cssmin": "^1.0.2",
13     "grunt-contrib-uglify": "^2.0.0",
14     "grunt-inline": "^0.3.6"
15   }
16 }
```

之后便可以使用这些插件了。以下是 Gruntfile.js 文件的内容：

```
01 module.exports = function (grunt) {
02     grunt.initConfig({
03         // 内联图片文件的配置
04         inline: {
05             page: {
06                 src: ['index.html']
07             }
08         },
09         // 压缩 JavaScript 文件的配置
10         uglify: {
11             dist: {
12                 files: {
13                     'dist/js/album.min.js': ['src/js/album.js']
14                 }
15             }
16         },
17         // 压缩 CSS 文件的配置
18         cssmin: {
19             dist: {
20                 files: {
21                     'dist/css/reset.min.css': ['src/css/reset.css'],
22                     'dist/css/album.min.css': ['src/css/album.css']
23                 }
24             }
25         },
26         // 合并多个文件的配置
27         concat: {
28             dist: {
29                 src: ['dist/css/reset.min.css', 'dist/css/album.min.css'],
30                 dest: 'dist/css/all.min.css'
31             }
32         }
33     });
34     // 加载这些任务
35     grunt.loadNpmTasks('grunt-inline');
36     grunt.loadNpmTasks('grunt-contrib-uglify');
37     grunt.loadNpmTasks('grunt-contrib-cssmin');
38     grunt.loadNpmTasks('grunt-contrib-concat');
39     // 默认执行的任务列表
```

```

41     grunt.registerTask('default', ['inline', 'uglify', 'cssmin', 'concat']);
42 };

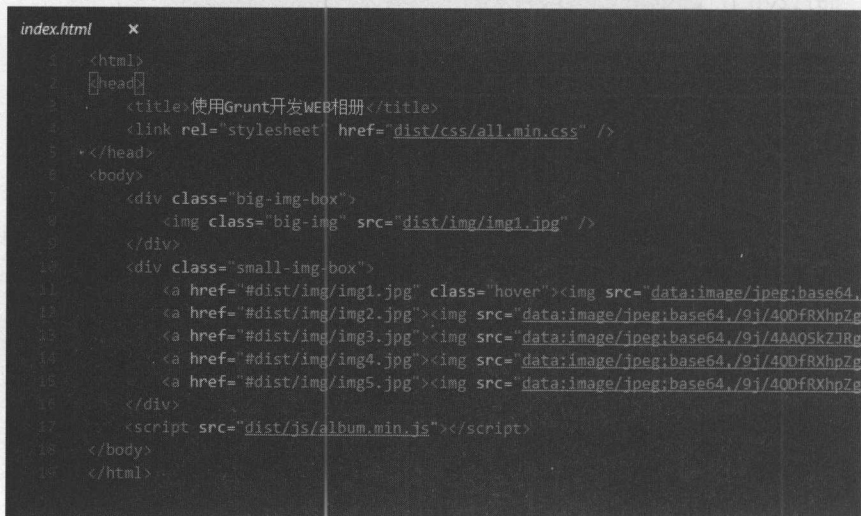
```

代码第 2~33 行，分别配置 4 个 Grunt 插件的运行数据。

代码第 35~38 行，使用 `grunt.loadNpmTasks` 方法将 4 个插件加载到项目中。

代码第 41 行，配置默认执行的任务列表（这里是 4 个任务全部执行）。

完成 `Gruntfile.js` 文件的配置之后，Grunt 方面的工作已基本上接近尾声。接下来是常规的项目开发阶段。完成开发后，只需要在终端运行 Grunt 命令，`Gruntfile.js` 文件里配置的 4 个任务就会自动执行。运行 Grunt 命令后的 `index.html` 文件内容如图 8.3 所示。



```

index.html  x
1  <html>
2  <head>
3  <title>使用Grunt开发WEB相册</title>
4  <link rel="stylesheet" href="dist/css/all.min.css" />
5  </head>
6  <body>
7  <div class="big-img-box">
8    
9  </div>
10 <div class="small-img-box">
11   <a href="#dist/img/img1.jpg" class="hover"></script>
18 </body>
19 </html>

```

图 8.3 运行 Grunt 命令后的文件截图

`index.html` 文件中只引用了一个合并后的 CSS 文件“`all.min.css`”和压缩后的 JavaScript 文件“`album.min.js`”。另外对于页面底部的缩略图，图片内容都转换成了 base64 的编码格式，直接内联到 HTML 文件中。这样做可以大大减少 HTTP 请求次数，以达到快速展示页面的目的。在浏览器中打开 `index.html` 文件的效果如图 8.4 所示。

注意：当 `Gruntfile` 文件中配置多个任务的时候，需根据任务之间的依赖关系决定任务运行的先后顺序。



图 8.4 打开 index.html 文件的效果图

8.3 使用 Gulp 构建一个 ECMAScript 6 和 Sass 应用

使用 Grunt 构建项目涉及磁盘操作, 构建效率较低, 因此, 基于流的 Gulp 应运而生。本节通过一个例子, 介绍如何利用 Gulp 构建一个 ECMAScript 6 和 Sass 应用。

8.3.1 安装和配置

运行 Gulp 需要 Node.js 环境, 请参看第二章内容搭建 Node.js 环境。使用 NPM 全局安装 Gulp, 命令如下:

```
npm install gulp-cli -g
```

然后, 在项目根目录下创建 package.json 文件, 命令如下:

```
npm init
```

根据引导配置项目信息, 然后安装 Gulp 依赖包, 命令如下:

```
npm install gulp -save-dev
```

在项目根目录下, 创建 gulpfile.js 文件, 内容如下:

```
var gulp = require("gulp");
```



```
gulp.task("default", function(){ // 定义名称为"default"的任务
  console.log("this is default task"); // 此处定义 default 任务处理过程。
});
```

和 Grunt 相似, Gulp 将构建过程拆解为一个独立的子任务, 使用 `gulp.task` 方法定义任务。通过命令提示符进入到项目目录, 用“gulp 任务名”执行任务, 实例命令如下:

```
gulp default
```

提示: 对于默认 (default) 任务, 可以省去任务名。

在创建任务的时候, 和 Grunt 相似, 可以指定任务的依赖项, 代码如下:

```
gulp.task("main", ["deps1", "deps2", ...], function(){
  // 相关执行
});
```

`gulp.task` 方法的第二个参数 (数组) 为“main”任务的依赖项。

项目中通常根据需求将构建过程拆解为多个小任务。下面介绍如何具体定义。

首先, 指定需要构建的内容, 并通过 Gulp 插件来完成构建, 最终输出到指定的目录。

采用 `gulp.src` 方法指定文件源, 代码如下:

```
gulp.src("src/**/*.js");
// 或者 对于多个目录下的源, 可以采用数组
gulp.src(["src/**/*.js", "theme/**/*.scss"]);
```

`gulp.src` 方法返回 Stream 对象, 可以通过 `pipe` 方法将内容传递给插件。所有插件都接受 `pipe` 传递过来的数据, 处理数据允许链式调用, 代码如下:

```
gulp.src("src/**/*.js").pipe(plugin1()).pipe(plugin2())...
```

构建完毕后, 需要采用 `gulp.dest` 方法将数据保存到文件中, 代码如下:

```
gulp.src("src/**/*.js").pipe(gulp.dest("dist")); // 读取 src 下的所有 js, 写入到 dist 目录下
```

Gulp 的每次操作都返回流对象, 所有操作在内存中进行, 不需要操作磁盘, 从而大幅提高了构建速度。

8.3.2 预处理任务

上一节中, 介绍了 Gulp 的安装、配置, 以及 Grunt 任务的定义和执行。本节将介绍编译 ECMAScript 6、Sass 和 CSS Sprite 任务。

“gulp-babel”插件可以将 ECMAScript 6 编译为 ECMAScript 5, 以便运行在不支持 ECMAScript 6 的浏览器上。首先安装该插件, 命令如下:

```
npm install gulp-babel -save-dev
npm install babel-preset-es2015 --save-dev
```

然后, 在 `gulpfile.js` 文件中创建任务, 代码如下:

```
var babel = require("gulp-babel"); // 引入 babel
gulp.task("compile-js", function(){
  gulp.src("src/**/*.js")          // 处理 src 下的所有 js 脚本
    .pipe(babel({                    // 调用 babel
      presets: ['es2015']            // 采用 es2015 预设插件, 将脚本编译为 ECMAScript 5
    }))
    .pipe(gulp.dest("dist/js"));     // 编译好的内容保存到 dist 目录下的 js 目录
});
```

Babel 可以将 JavaScript 文件, 甚至 React 的 JSX 文件编译为标准的 JavaScript 文件。Babel 官方提供的预设插件 (presets) 让用户能够更简单地使用 Babel。presets 是针对特定编译条件预设的一组插件集合。如本实例中采用的 es2015 预设插件包含插件有 “check-es2015-constants”、 “transform-es2015-arrow-functions” 等。

注意: Babel 只是做了语法层次的转换, 并不会增加 API 的支持。对于 class 关键字定义的类, Babel 会将其转化为通过 prototype 定义的对象。而对于 ECMAScript 6 的 Promise 对象, Babel 不会做任何处理, 因此需要通过 polyfill 来对浏览器不支持的 API 进行扩展。如 “es6-promise” 使得浏览器支持 Promise 对象。

在实际的项目中, 可以在根目录下创建的 “.babelrc” 文件中配置 Babel, 代码如下:

```
{
  "presets": ["es2015"]
}
```

编译 Sass 文件可以采用 gulp-sass 插件, 安装命令如下:

```
npm install gulp-sass -save-dev
```

然后, 在 `gulpfile.js` 文件中, 增加任务执行 Sass 编译, 代码如下:

```
var sass = require("gulp-sass"); // 引入 sass 插件
gulp.task("compile-sass", function(){ // 定义编译 sass 的任务
  gulp.src("theme/**/*.scss")        // 处理 theme 下所有的 sass 文件
    .pipe(sass().on('error', sass.logError)) // 采用 sass 插件编译, 并处理错误
});
```

```
.pipe(gulp.dest("dist/css")); // 编译好的内容输出到 dist 目录下的 css 目录
});
```

在项目中，为了优化加载性能，需要将小图片合成一张大图，也就是所谓的“CSS Sprites”。该功能使用 `gulp.spritesmith` 插件来实现，安装命令如下：

```
npm install gulp.spritesmith --save-dev
```

然后在 `gulpfile.js` 文件中建立任务，代码如下：

```
var spritesmith = require('gulp.spritesmith'); // 引入 sprite 插件
gulp.task("sprite", function () { // 定义任务
  gulp.src("theme/images/**/*.png") // 处理 theme 目录下的 png 文件
    .pipe(spritesmith({ // 调用插件合并图片
      imgName: 'sprite.png', // 输出合成的图片名称
      cssName: 'sprite.css' // 输出对应的 css 文件
    }))
    .pipe(gulp.dest("dist")); // 输出到 dist 目录
});
```

本节介绍了三个预处理工具：`gulp-babel`、`gulp-sass` 和 `gulp.spritesmith`。

8.3.3 实战演练：采用 ECMAScript 6 开发一个 Markdown 编辑器

本节将采用 ECMAScript 6 开发一个 Markdown 编辑器，界面效果如图 8.5 所示。

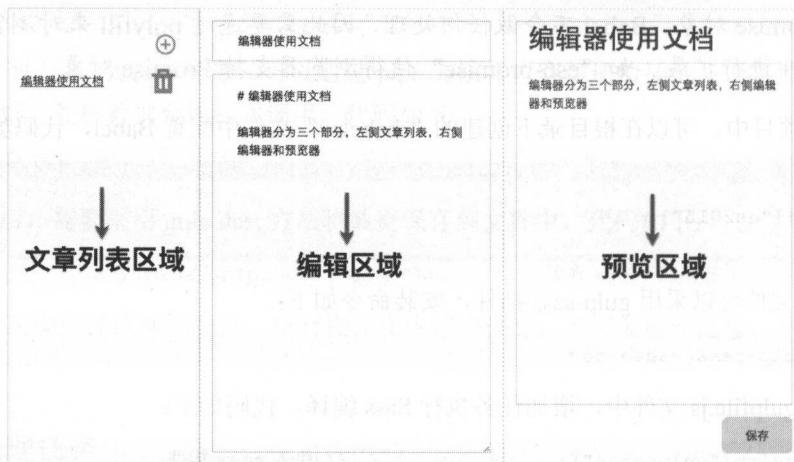


图 8.5 Markdown 编辑器效果图

编辑器包含文章列表展示、新增、删除、读取、编辑、预览和保存功能。

本例中，使用了开源的 Markdown 解析器 showdown 类库，用以将 Markdown 解析为 HTML，项目地址为 <https://github.com/showdownjs/showdown>。

为了更方便地进行演示，实例中的存储部分通过 HTML 5 的 indexedDB 进行保存。

HTML 部分，代码如下：

```
01 <body>
02   <nav><!-- 省略部分代码 --></nav>           <!-- 导航部分 -->
03   <main>                                       <!--编辑部分 -->
04     <div class="editor">
05       <!--省略部分代码-->
06       <div class="md-editor">                 <!--markdown 编辑区域 -->
07         <textarea placeholder="请填写 Markdown" id="tbMarkdown"></textarea>
08       </div>
09     </div>
10     <div class="preview-container"><!--省略部分代码 --></div>   <!--预览部分 -->
11   </main>
12 </body>

2. Sass:
01 body{
02   display: flex;           // 采用 flex 布局
03   height: 90vh;           // 高度为 view-port 高度的 90%
04 }
05 nav{
06   width: 200px;           // 导航菜单宽度固定
07 }
08 main{
09   flex: 1;                // 主体内容宽度自适应
10 }
```

JavaScript 部分，代码如下：

```
01 const $markdown = document.getElementById("tbMarkdown") // 获取 markdown 输入框
02 const $preview = document.getElementById("preview")     // 获取 preview 元素
03 const mdConverter = new showdown.Converter()            // 调用 showdown
04 $markdown.addEventListener("keyup", () => {
05   renderPreview()                                       // 输入字符时进行渲染
06 })
07 const renderPreview = () => {
08   $preview.innerHTML = convertMarkdown($markdown.value) // 解析 markdown
09 }
```


文章保存在 indexedDB 中, 采用 ECMAScript 6 的 Promise 实现异步调用。在操作 indexedDB 之前需先打开 DB 连接, 代码如下:

```
01 const openDB = (version = 1) => {
02     return new Promise((resolve, reject) => {           // 封装 Promise 对象
03         const request = indexedDB.open(DB_NAME, version) // 打开 DB 连接
04         request.addEventListener("error", (e) => {
05             reject(e.target.error)
06         })
07         request.addEventListener("success", (e) => {
08             resolve(e.target.result)
09         })
10         request.addEventListener("upgradeneeded", (e) => {
11             const db = e.target.result
12             if (!db.objectStoreNames.contains(STORE_NAME)) { // 定义数据表的主键
13                 db.createObjectStore(STORE_NAME, {
14                     keyPath: "id",
15                     autoIncrement: true
16                 })
17             }
18         });
19     });
20 }
```

注意: indexedDB 通过版本来管理数据库的结构, 因此必须在 upgradeneeded 事件中修改数据库或者对象存储的代码。当版本发生变化时, 此事件被触发。其中, 版本号只能增加, 不能减少。

indexedDB 的接口支持基于事件的异步调用, 可以将其封装为 Promise 对象, 代码如下:

```
const wrapPromise = (fn) => {
    return new Promise((resolve, reject) => {           // 返回 Promise 对象
        const request = fn()                          // 执行回调方法, 得到异步执行对象
        request.addEventListener("error", (e) => {     // 绑定错误事件
            reject(e.target.error)
        })
        request.addEventListener("success", (e) => { // 绑定成功事件
            resolve(e.target.result)
        })
    })
}
```

调用方式，代码如下：

```
01 const getStore = (mode = "readonly") => {
02     return openDB().then((db) => {          // 打开 DB 连接
03         // 建立事务，并返回存储对象。
04         return db.transaction(STORE_NAME, mode).objectStore(STORE_NAME)
05     })
06 }
07 const add = (data) => {                    // 添加数据
08     return getStore('readwrite').then((store) => { // 调用 getStore 方法，获取 store 对象
09         return wrapPromise(()=>{          // 调用 wrapPromise 方法，返回 Promise 对象
10             return store.add(data); // 执行数据操作，将其返回给 wrapPromise
11         })
12     });
13 }
```

本例介绍了如何采用 ECMAScript 6 和 Sass 构建一个 Markdown 编辑器。受篇幅所限，具体代码请参考本书附带源码。

8.3.4 代码检查任务

在项目的开发过程中，统一的代码风格对于项目的可协作性以及可维护性来说相当重要，因此可以采用一些插件来进行代码风格的检查。

本例中的源文件包含两类：Sass 文件和采用 ECMAScript 6 规范的 JavaScript 文件。在 Gulp 中，采用 gulp-eslint 和 gulp-sass-lint 插件来分别进行检测。

(1) 代码检查任务中，采用开源的 JavaScript 验证工具 ESLint 进行处理，执行任务前需要先全局安装 ESLint，命令如下：

```
npm install eslint -g
```

(2) 在项目目录下执行 ESLint 初始化命令，创建 ESLint 的配置文件，命令如下：

```
eslint -init
```

(3) 进行 ESLint 项目初始化向导，首先选择配置方式，这里选择“Use a popular style guide”，如图 8.6 所示。

(4) 选择预设模式，这里选择“Standard”，如图 8.7 所示。

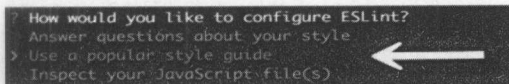


图 8.6 选择配置方式

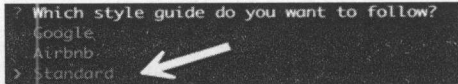


图 8.7 选择预设模式

(5) 选择配置内容的存储文件类型，如图 8.8 所示。

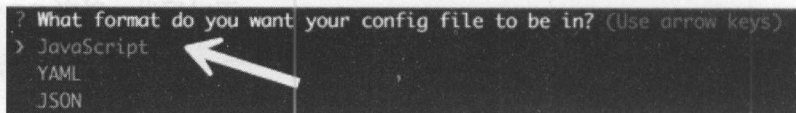


图 8.8 选择配置内容的存储文件类型

至此 ESLint 项目配置结束，目录中产生一个名为“.eslintrc”的文件。接下来介绍插件 `gulp-eslint` 的安装使用。首先在项目中安装插件，命令如下：

```
npm install gulp-eslint -save-dev
```

在 `gulpfile.js` 文件中，添加任务对 JavaScript 代码进行检查：

```
var eslint = require("gulp-eslint");           // 引入 eslint 插件
gulp.task("eslint", function(){               // 定义代码检查任务
  gulp.src("src/**/*.js")                     // 对 src 下的所有 js 文件执行代码检查
    .pipe(eslint({                             // 执行检查
      useEslintrc: true                       // 采用 .eslintrc 配置文件
    }))
    .pipe(eslint.format())                     // 输出检查结果
    .pipe(eslint.failAfterError())             // 当代码检查失败时，终止 gulp 任务
});
```

上述例子简单地介绍了如何在 Gulp 中采用 ESLint 插件执行代码检查。更多 ESLint 的配置项和插件使用文档请参看 <http://eslint.org/>。

注意：ESLint 提供了两种方式终止任务：`eslint.failAfterError` 和 `eslint.failOnError`。前者会等到所有检查执行完才终止，而后者则在出错时立即终止。

接下来，介绍采用 `gulp-sass-lint` 插件对 Sass 文件进行代码检查。首先，安装插件：

```
npm install gulp-sass-lint --save-dev
```

在 `gulpfile.js` 中添加任务对 Sass 文件进行代码检查，代码如下：

```
var sasslint = require("gulp-sass-lint");      // 引入 sasslint 插件
gulp.task("sasslint", function(){             // 定义 sasslint 任务
  gulp.src("theme/**/*.scss")
    .pipe(sasslint())                          // 采用 sasslint 插件，执行代码检查
    .pipe(sasslint.format())                   // 输出检查结果
    .pipe(sasslint.failOnError())              // 当出错时，终止 gulp 任务
});
```

本节简要地介绍了如何采用 Gulp 插件对代码进行检查。关于 ESLint 和 SassLint 的进一步用法请参考对应文档。

8.3.5 代码合并、压缩、重命名任务

上一节介绍了代码检查任务，在代码编写的时候，通过代码检查任务，可以有效地统一代码风格，尽早发现代码中的错误。在前端性能优化阶段，通常需要对代码进行合并，以减少请求数；对代码进行压缩，以减少传输体积。某些时候还需要重命名某些代码。

开发时为了使代码便于阅读，通常需要根据功能模块，将代码保存在不同的文件中，本例中的文件结构如图 8.9 所示。

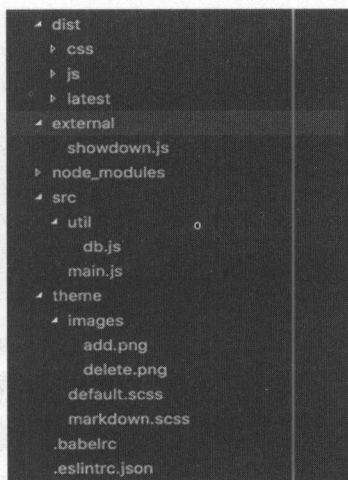


图 8.9 文件结构图

在 Gulp 中，采用 gulp-concat 插件将各个独立的文件进行合并。安装命令如下：

```
npm install --save-dev gulp-concat
```

在 gulpfile.js 中创建合并任务，代码如下：

```
var concat = require("gulp-concat"); // 引入 concat 插件
gulp.task("concat", ["compile-js", "compile-sass", "sprite", "copy-images"], function () {
  gulp.src("dist/css/*.css") // 对由 Sass 编译的 css 文件合并
    .pipe(concat("style.css")) // 合并为 style.css 文件
    .pipe(gulp.dest("dist/latest")); // 保存合并的文件
  gulp.src("dist/js/**/*.js") // 对 ECMAScript6 编译的 Javascript 文件合并
    .pipe(concat("app.js")) // 合并成 app.js 文件
});
```



```
.pipe(gulp.dest("dist/latest")); // 保存文件
});
```

在定义合并任务时，将预处理任务定义在此任务的依赖项里：`compile-js`、`compile-sass`、`sprite`、`copy-images`。

通常情况下，项目上线时需对合并后的代码进行压缩。`JavaScript` 代码压缩采用“`gulp-uglify`”插件，`CSS` 代码压缩可以采用“`gulp-clean-css`”插件。首先安装对应的插件：

```
npm install gulp-uglify gulp-clean-css -save-dev
```

在 `gulpfile.js` 中，创建代码压缩的任务：

```
01 var uglify = require("gulp-uglify"); // 引入 uglify 插件
02 var cssMinify = require("gulp-clean-css"); // 引入 clean-css 插件
03 gulp.task("minify", ["concat"], function(){ // 定义 minify 任务
04     gulp.src("dist/latest/app.js") // 找到 Javascript 文件
05     .pipe(uglify()) // 采用 uglify 插件进行压缩
06     .pipe(gulp.dest("dist/latest")); // 保存文件
07     gulp.src("dist/latest/style.css") // 找到 CSS 文件
08     .pipe(cssMinify()) // 采用 clean-css 插件进行压缩
09     .pipe(gulp.dest("dist/latest")); // 保存文件
10 });
```

执行 `minify` 任务后，在“`dist/latest`”目录下，`app.js` 和 `style.css` 都变成了最小化的文件。接下来对生成的压缩文件重命名。该功能使用插件“`gulp-rename`”，安装命令如下：

```
npm install gulp-rename -save-dev
```

将上文的代码稍作调整，完成文件重命名功能，代码如下：

```
01 gulp.src("dist/latest/app.js")
02     .pipe(uglify())
03     .pipe(rename(function(path){ // 调用 rename 插件
04         path.basename += ".min" // 修改文件名，在文件名后增加 ".min"
05     })))
06     .pipe(gulp.dest("dist/latest"));
```

本节介绍了对预处理后的文件进行合并、压缩和重命名。至此，采用 `Gulp` 构建出来的代码就可以发布到线上了。

8.3.6 监听文件变化自动构建

前例介绍了如何采用 `Gulp` 构建使用 `ECMAScript` 和 `Sass` 开发的项目。开发过程中还可以通过

一些插件来加速开发调试。本节介绍如何监听文件变化自动构建并在修改后进行自动刷新。

使用 Gulp 的 watch 方法来监听文件变化自动构建，代码如下：

```
gulp.task("watch", ["concat"], function () {
  // 当JS和Sass文件修改时，自动执行"concat"任务
  gulp.watch(["src/**/*.js", "theme/**/*.scss"], ["concat"]);
});
```

这样，当 src 目录下的 Sass 文件和 JavaScript 文件有所更改时，concat 任务自动执行，构建出合并完成的 JavaScript 和 CSS。

可以采用 browser-sync 工具来实现浏览器自动刷新，安装命令如下：

```
npm install browser-sync --save-dev
```

首先定义 browser-sync 任务，启动由 browser-sync 提供的 Web 服务，命令如下：

```
var browserSync = require("browser-sync");
var reload = browserSync.reload;
gulp.task("browser-sync", function () { // 定义 browser-sync 任务。
  browserSync({
    server: './' // Web Server 的根目录为当前目录
  });
});
```

执行 browser-sync 任务后，浏览器将自动打开并访问根目录下的默认文件。接下来需要修改 concat 任务，使得在监听文件变化时同步刷新浏览器，代码如下：

```
gulp.task("concat", ["compile-js", "compile-sass", "sprite", "copy-images"], function () {
  gulp.src("dist/css/*.css")
    .pipe(concat("style.css"))
    .pipe(gulp.dest("dist/latest"))
    .pipe(reload({ // 执行前面定义的 browserSync.reload 刷新浏览器
      stream: true
    }));
  <!--省略，参考书籍源码 --!>
});
```

本节通过具体的例子介绍了如何采用 Gulp 构建项目；通过代码检查，规范代码书写和提前发现代码错误；通过 browser-sync 同步浏览器效果，提升开发效率。相比 Grunt 来说，Gulp 采用了 Stream 的方式，明显提升了构建效率。

提示：Grunt 的出现早于 Gulp，也是基于 Node.js 的项目构建工具。官网地址为：
<http://gruntjs.com/>。

8.4 实战演练：使用 Webpack 构建一个 React 应用

Webpack 是一个模块加载器兼打包工具，能把各种资源，例如脚本（JavaScript、TypeScript、JSX）、样式（CSS、Less、Sass）、图片（png、jpg、gif）等都作为模块来处理。目前，主流前端开发框架（如 React、Vue.js）的官方实例项目均使用 Webpack 作为工程化工具。

8.4.1 安装和配置

本节会创建一个简单的 React 项目，并使用 Webpack 构建开发环境以及打包生产环境所需的静态资源。项目目录结构如图 8.10 所示。

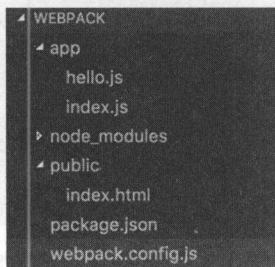


图 8.10 项目目录结构

app 文件夹存储打包前的静态文件，里面的文件会被 Webpack 打包，打包后的文件会存储到 public 文件夹。

index.js 文件是入口脚本文件，内容如下：

```

import React, { Component } from 'react';           // 引入 react 包
import ReactDOM from 'react-dom';                   // 引入 react-dom 包
import Hello from './hello';                         // 引入依赖的 Hello 组件
class App extends Component {                       // 定义入口组件
  render() {                                         // 绘制组件
    return (
      <div>
        <Hello />                                // 使用 Hello 组件
      </div>
    )
  }
}
  
```

```

    }
  }
  ReactDOM.render(                                // 绘制组件到页面
    <App />,
    document.getElementById('root')
  );

```

hello.js 文件定义了一个 React 组件，被 index.js 文件使用，代码如下：

```

import React, { Component } from 'react';          // 引入 react 包
export default class Hello extends Component {      // 定义 Hello 组件并导出
  render() {                                         // 绘制组件
    return (
      <div>
        Hello World
      </div>
    )
  }
}

```

注意：React 的详细使用会在后续章节中介绍，此处只是简单使用。

public 文件夹存储需要被浏览器访问到的文件，打包后生成的入口脚本文件也会存储在这个文件夹当中。

index.html 是入口页面文件，内容如下：

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Webpack Sample Project</title>
  </head>
  <body>
    <!-- React 组件绘制的 DOM 会作为子元素插入 -->
    <div id='root'>
    </div>
    <!-- 引用入口脚本文件 -->
    <script src="./bundle.js"></script>
  </body>
</html>

```

package.json 文件是标准的 NPM 说明文件，这个文件可以通过 NPM 命令生成。打开命令行

窗口, 输入如下命令:

```
npm init
```

控制台会逐步提示用户输入创建 `package.json` 所需要的相关信息, 例如项目名称、版本号等。
`package.json` 文件内容如下:

```
{
  "name": "webpack-demo",           // 项目名称
  "version": "1.0.0",               // 项目版本
  "description": "webpack demo",    // 项目描述
  "main": "index.js",              // 项目入口文件
  "scripts": {                     // 项目支持的脚本命令
    "start": "webpack-dev-server"  // start 命令, 用于启动开发环境
    "build": "webpack "            // build 命令, 用于打包生产环境静态资源
  },
  "author": "",                     // 项目作者
  "license": "ISC",                 // 项目 license
  "dependencies": {                 // 项目运行时依赖的包
    "react": "^15.4.1",
    "react-dom": "^15.4.1"
  },
  "devDependencies": {              // 项目开发时依赖的包
    "babel-core": "^6.21.0",
    "babel-loader": "^6.2.10",
    "babel-preset-latest": "^6.16.0",
    "babel-preset-react": "^6.16.0",
    "webpack": "^1.14.0",
    "webpack-dev-server": "^1.16.2" // 提供 Web 服务
  }
}
```

`webpack.config.js` 是 Webpack 配置文件, 告诉 Webpack 需要做什么, Webpack 运行时会读取此文件, 内容如下:

```
var webpack = require('webpack'); // 引入 webpack 包
module.exports = {                 // 导出配置
  entry: __dirname + "/app/index.js", // 入口文件
  output: {
    path: __dirname + "/public",      // 输出路径
    filename: "bundle.js"             // 输出文件名
  },
  module: {
```

```

    loaders: [                                     // 加载器列表
      {
        test: /\.js$/,                             // 处理后缀名为 js 的文件
        exclude: /node_modules/,                   // 需要忽略的文件夹
        loader: 'babel',                             // 加载器
        query: {
          presets: [                                 // 插件集列表
            'es2015',                               // 处理 es2015 的 Babel 插件
            'react'                                 // 处理 react 的 Babel 插件
          ]
        }
      }
    ],
    devServer: {                                    // Web Server 配置
      contentBase: "./public",                       // 暴露给 Web Server 的文件夹
      inline: true                                   // Web Server 运行模式
    }
  }
}

```

项目配置完成之后，安装项目所需 NPM 依赖包。打开命令行窗口，运行如下命令：

```
npm install
```

npm 包安装成功之后，启动项目。打开命令行窗口，运行如下命令：

```
npm run start
```

项目成功启动之后，可以打开浏览器，输入地址 <http://localhost:8080/index.html> 访问页面。

注意：开发环境下，打包生成的静态资源 `bundle.js` 存储在内存里面，不会输出到磁盘，在 `public` 文件夹下无法看到 `bundle.js` 文件。

如果想生成生产环境所需静态资源，打开命令行窗口，输入如下命令：

```
npm run build
```

命令运行成功后，`public` 文件夹下会生成打包后的静态资源 `bundle.js`。

8.4.2 常用的加载器和插件

加载器的作用是对项目中的源文件进行格式转换，以函数的形式存在，接收源文件作为输入参数，输出转换后的资源文件。

上一节的例子中使用了 **babel-loader** 加载器, **babel-loader** 加载器的作用是将使用 ECMAScript 6 语法编写的脚本文件转换成 ECMAScript 5, 并对 React 的 JSX 语法进行转换。webpack.config.js 文件相关的配置代码如下:

```
01 loaders: [                                // 加载器列表
02     {
03         test: /\.js$/,                    // 处理后缀名为 js 的文件
04         exclude: /node_modules/,         // 需要忽略的文件夹
05         loader: 'babel',                 // 加载器
06         query: {
07             presets: [                    // 插件集列表
08                 'es2015',                 // 处理 es2015 的 Babel 插件
09                 'react'                  // 处理 react 的 Babel 插件
10             ]
11         }
12     }
13 ]
```

代码第 03 行配置当前加载器需要处理的文件, 通过对文件完整路径进行正则匹配确定。这里配置的是处理所有以“js”为后缀名的文件。

代码第 04 行配置当前加载器不需要处理的文件, 通过对文件完整路径进行正则匹配确定。这里配置的是所有在 **node_modules** 文件夹中的文件都不需要。

代码第 05 行配置当前加载器名称, 这里配置的是 **babel-loader** 加载器。加载器以名字标识, 可以省略“-loader”。

代码第 06~11 行配置当前加载器需要的参数。这里配置 **babel-loader** 需要使用 **es2015** 和 **react** 插件集。

项目中一般会使用 CSS 预编译器, 比如 **Sass**。在使用 **Webpack** 打包的时候需要处理 **Sass** 文件, 这时可以使用 **sass-loader** 加载器。webpack.config.js 文件相关的配置如下:

```
loaders: [{                                // 加载器列表
    test: /\.scss$/,                      // 处理后缀名为 scss 的文件
    loader: 'sass',                       // 加载器
  }]
```

sass-loader 加载器会将 **Sass** 文件编译成 **CSS** 文件, 但是 **CSS** 文件里面包含的“@import”和“url(...)”语法还需要进行进一步处理, 这时候可以使用 **css-loader** 加载器。webpack.config.js 文件相关的配置代码如下:


```

loaders: [{                                // 加载器列表
  test: /\.scss$/,                        // 处理后缀名为 scss 的文件
  loader: 'css!sass',                    // 加载器
}]

```

为了处理浏览器兼容性,在书写 CSS 代码的时候,需要给某些 CSS 属性添加浏览器前缀,这个工作也可以通过 Webpack 加载器完成。将 postcss-loader 加载器添加到 sass-loader 之后,css-loader 之前,webpack.config.js 文件相关的配置代码如下:

```

module: {
  loaders: [{                                // 加载器列表
    test: /\.scss$/,                        // 处理后缀名为 scss 的文件
    loader: 'css!postcss!sass' ,           // 加载器
  ]},
  postcss: function () {
    return [autoprefixer];                // 自动添加前缀
  },
}

```

使用 css-loader 加载器处理 CSS 文件之后,处理完的 CSS 文件已经可以被页面使用了。可以使用 style-loader 加载器将 CSS 作为内联样式嵌入到页面。webpack.config.js 文件相关的配置代码如下:

```

module: {
  loaders: [{                                // 加载器列表
    test: /\.scss$/,                        // 处理后缀名为 scss 的文件
    loader: 'style!css!postcss!sass',       // 加载器
  ]},
  postcss: function () {
    return [autoprefixer];                // 自动添加前缀
  },
}

```

这种情况下,CSS 样式包含在入口脚本文件里面。当入口脚本文件被加载到页面时会自动将 CSS 作为内联样式嵌入到页面 Style 标签中。

还有一种使用处理后的 CSS 文件的方式是将 CSS 文件从入口脚本文件抽离出来,作为单独的入口样式文件加入页面。webpack.config.js 文件相关的配置代码如下:

```

module: {
  loaders: [{                                // 加载器列表
    test: /\.scss$/,                        // 处理后缀名为 scss 的文件

```



```

    loader: ExtractTextPlugin.extract('css!postcss!sass'), // 加载器
  },
  postcss: function () {
    return [autoprefixer]; // 自动添加前缀
  },
  plugins: [
    new ExtractTextPlugin("bundle.css") // 输出的 CSS 文件名
  ]
}

```

这种情况下, CSS 将从入口脚本文件中抽离出来, 作为一个单独的入口样式文件, 并在入口页面文件里直接引用。相关代码如下:

```

<html>
  <head>
    <!-- 引用入口样式文件 -->
    <link rel="stylesheet" href=". /bundle.css">
  </head>
  <body>
    <!-- React 组件绘制的 DOM 会作为子元素插入 -->
    <div id='root'>
    </div>
    <!-- 引用入口脚本文件 -->
    <script src=". /bundle.js"></script>
  </body>
</html>

```

将 CSS 从脚本里面抽离出来是使用 ExtractTextPlugin 插件来完成的。Webpack 插件是一种比加载器更强大的机制, 可以在 Webpack 编译的各阶段注册钩子, 并且访问 Webpack 提供的底层 API。

打包的过程通常包含代码压缩的需求。HTML 文件的压缩通过 html-minify-loader 加载器来完成, 脚本文件的压缩通过 UglifyJsPlugin 插件来完成。相关代码如下:

```

module: {
  loaders: [{ // 加载器列表
    test: /\.html$/, // 处理后缀名为 html 的文件
    loader: 'html?minimize=false!html-minify', // 加载器
  }],
  'html-minify-loader': {
    comments: true // 保留注释
  },
}

```

```

plugins: [
  new webpack.optimize.UglifyJsPlugin()           // 压缩代码
]
}

```

DefinePlugin 插件用来定义一些全局变量，这些变量可以在模块当中直接使用，比如通常定义一个变量来标识当前是开发环境还是生产环境。相关代码如下：

```

module: {
  plugins: [
    new webpack.DefinePlugin({                     // 定义全局变量
      'process.env.NODE_ENV': '"production"'
    })
  ]
}

```

定义的变量可以在模块中直接使用，比如入口脚本文件 index.js。使用方法如下：

```

if (process.env.NODE_ENV === 'production') {
  // 如果是生产环境
}
else {
  // 如果是开发环境
}

```

8.4.3 缓存控制

为了提高页面加载速度以及减轻服务器压力，通常服务端返回响应时会告诉浏览器缓存静态资源的方式。当浏览器第一次访问资源之后，会根据服务器在 HTTP 协议中设定的时间缓存资源在客户端本地，在缓存有效期内，后续对该资源的访问都会直接从缓存中读取，不会再向服务器发起请求。沪江网校列表页其中一个脚本文件的网络请求如图 8.11 所示。

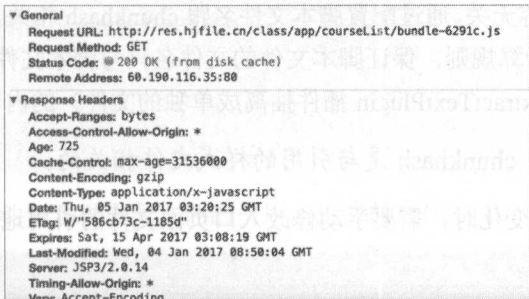


图 8.11 缓存的脚本

提示：沪江网校列表页地址为 <http://class.hujiang.com/category>。

从“Request URL”可以看出，这次请求的静态资源是 bundle-6291c.js。“Cache-Control”表示服务器告诉浏览器缓存此资源 31536000 秒。“Status Code”表示此次请求没有发送给服务器端，是直接从浏览器缓存中读取。

静态资源缓存减少了浏览器端请求服务器资源的次数，同时也带来了新的问题：一旦文件内容有所更新，继续访问原有的 URL 地址，用户获取的资源文件仍然是先前版本。所以，解决方案是将文件内容生成的哈希值作为文件名的一部分，一旦文件内容改变，生成的文件名就会改变，即浏览器会请求新的 URL 地址，完成新版本文件的上线。

Webpack 提供了根据文件内容确定输出文件名字的功能，webpack.config.js 文件相关的配置代码如下：

```
module: {
  output: {
    path: __dirname + "/public",           // 输出文件的路径
    filename: "bundle-[chunkhash].js"      // 文件名字跟内容哈希值有关
  },
  loaders: [{                             // 加载器列表
    test: /\.css$/,                       // 处理后缀名为 css 的文件
    loader: ExtractTextPlugin.extract('css'), // CSS 抽离成单独文件
  ]},
  plugins: [
    new WebpackMd5Hash(),                 // 分离样式文件与脚本文件哈希值
    new ExtractTextPlugin("bundle-[contenthash].css") // 输出的 CSS 文件名跟内容相关
  ]
}
```

通过配置样式文件名跟 contenthash 相关，保证样式文件的文件名只与样式文件本身的内容相关，与引用其脚本文件内容无关。通过配置脚本文件名跟 chunkhash 相关，并引入 WebpackMd5Hash 插件修改 chunkhash 的计算规则，保证脚本文件的文件名只与脚本文件本身的内容相关，与其引用的样式文件（已经被 ExtractTextPlugin 插件抽离成单独的文件）的内容无关。

注意：默认情况下，chunkhash 是与引用的样式文件相关的。

当资源文件名字发生变化时，需要手动修改入口页面文件的引用地址，代码如下：

```
<html>
  <head>
    <!-- 样式资源内容变化时，手动修改引用的入口样式文件名字 -->
```



```

    <link rel="stylesheet" href=". /bundle-8b11f5614427e1a8eccae316ea741d29.css">
  </head>
  <body>
    <!-- React 组件绘制的 DOM 会作为子元素插入 -->
    <div id='root'>
    </div>
    <!-- 脚本资源内容变化时，手动修改引用的入口脚本文件名字 -->
    <script type="text/javascript" src=". /bundle-3e85e6ba4ce17877a07d.js"></script>
  </body>
</html>

```

每次手动修改入口页面文件还是比较麻烦的，因此，可以通过 `HtmlWebpackPlugin` 插件来实现入口文件引用地址的自动变化。`webpack.config.js` 文件相关的配置代码如下：

```

{
  plugins: [
    new HtmlWebpackPlugin({
      template: 'app/index.html' // 入口页面模板
    })
  ]
}

```

Webpack 读取 `app` 文件夹下的 `index.html` 文件作为模板文件，然后生成入口文件 `index.html` 到 `public` 文件下，模板文件代码如下：

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Webpack Sample Project</title>
  </head>
  <body>
    <!-- React 组件绘制的 DOM 会作为子元素插入 -->
    <div id='root'>
    </div>
  </body>
</html>

```

生成的入口文件会自动包含对资源文件的引用，生成的代码如下：

```

<!DOCTYPE html>
<html>
  <head>

```



```

<meta charset="utf-8">
<title>Webpack Sample Project</title>
<!-- Webpack 自动添加对样式文件的引用 -->
<link rel="stylesheet" href=". /bundle-8b11f5614427e1a8eccae316ea741d29.css">
</head>
<body>
  <!-- React 组件绘制的 DOM 会作为子元素插入 -->
  <div id='root'>
    </div>
  <!-- Webpack 自动添加对脚本文件的引用 -->
  <script type="text/javascript" src=". /bundle-3e85e6ba4ce17877a07d.js"></script>
</body>
</html>

```

8.4.4 简化模块引用

Webpack 引用模块时，指定模块路径的方式分三种：相对路径、绝对路径和模块路径，另外还有多种简化模块引用路径的方式。

下面使用一个实例演示 Webpack 跟模块引用相关的配置，实例目录结构如图 8.12 所示。

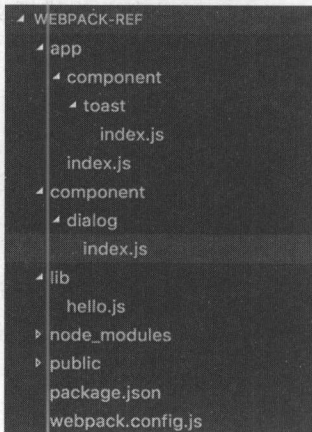


图 8.12 项目目录结构

文件“/app/index.js”使用相对路径引用文件“/lib/hello.js”，代码如下：

```
import hello from "../lib/hello.js"
```

文件“/app/index.js”使用绝对路径引入文件“/lib/hello.js”，代码如下：

// 此处路径应替换为本地真实磁盘路径

```
import hello from "/Users/YourUserName/Documents/source code/webpack-ref/lib/hello.js"
```

文件“/app/index.js”使用模块路径引入文件“/lib/hello.js”，代码如下：

```
import hello from "hello.js"
```

默认情况下，模块加载的查找路径是全局 NPM 包所在目录和本地 NPM 包所在目录（即项目的 node_modules 目录），此处需要配置 Webpack 将特定目录当作模块查找路径。webpack.config.js 文件相关的配置代码如下：

```
{
  resolve: {
    root: [                                // 配置绝对查找路径
      path.resolve("./lib")              // 指定 lib 目录为模块查找路径
    ]
  }
}
```

注意：root 配置的路径必须是绝对路径，此处用 path.resolve 获取绝对路径。

配置模块加载的查找路径时，除了可以配置绝对路径，还可以配置相对路径。假设文件“/app/index.js”需要加载某个文件，这个文件既有可能存在于目录“/app/component”下，也有可能存在于目录“/component”下，这个时候可以配置相对查找路径。

文件“/app/index.js”使用模块路径引入文件“/component/dialog/index.js”和“/app/component/toast/index.js”，代码如下：

```
import toast from "toast/index.js";
import dialog from "dialog/index.js";
webpack.config.js 文件相关配置如下：
```

```
{
  resolve: {
    modulesDirectories: [                 // 配置相对查找路径
      "web_modules",                     // 默认查找路径，自定义路径的时候不能落掉默认的
      "node_modules",                   // 默认查找路径，自定义路径的时候不能落掉默认的
      "component",                       // 设置 component 为相对查找路径
    ]
  }
}
```

Webpack 会先在文件“/app/index.js”的当前目录下查找是否存在目录 component，如果目录“/app/component”存在，会接着查找文件“/app/component/toast/index.js”是否存在，该文件存在，

查找结束。但如果文件“/app/component/dialog/index.js”不存在，Webpack 会重新到文件“/app/index.js”的父目录下查找是否存在目录 component，如果目录“/component”存在，会接着查找文件“/component/toast/index.js”是否存在，该文件存在，查找结束。如果文件还不存在，会继续往上层目录查找，直至磁盘根目录。

在引用模块文件时，文件的后缀名可以省略，Webpack 提供补全机制。比如后缀“.js”就可以省略，Webpack 默认会进行补齐，所以文件“/app/index.js”使用模块路径引入文件“/component/dialog/index.js”可以进行简化，代码如下：

```
import toast from "dialog/index";
Webpack 支持自定义补全后缀，webpack.config.js 文件相关配置如下：
{
  resolve: {
    extension: [                                // 自定义需要自动补全的后缀
      "",                                       // 不补全的情况
      ".js",                                  // js 后缀是默认情况，自定义时加上防止覆盖
      ".scss"                                // 自定义补全 scss 后缀
    ]
  }
}
```

注意：自定义配置会覆盖默认配置，重新定义默认配置。

在引用模块文件时，可以把具体文件名省略，只指定到目录。如果目录下存在 index.js 文件，Webpack 会默认加载该文件。文件“/app/index.js”使用模块路径引入文件“/component/dialog/index.js”可以据此简化，代码如下：

```
import toast from "dialog";
```

如果想实现默认加载某个指定的文件，需要在目录下创建 package.json 文件，package.json 文件相关的配置代码如下：

```
{
  main: "entry.js"                            // 设置 entry.js 为默认加载文件
}
```

8.4.5 异步模块加载

随着页面功能越来越复杂，资源文件也会变大，尤其是对于单页应用来说。单页应用默认所有资源都会被编译到一个统一的文件中，因此过大的资源文件会加重网络传输负担，影响首屏加载速度。通常情况下，不是所有资源都会被立刻使用，这就可以将资源按照业务需求打包成不同

的文件，最开始的时候只加载最为必要的资源，其他则等需要时再进行加载。

使用 Webpack，可以在代码中指定将哪些资源模块单独打包成资源包，并且在需要时自动加载所需资源包。下面用一个简单的例子演示 Webpack 异步资源加载功能，项目目录结构如图 8.13 所示。

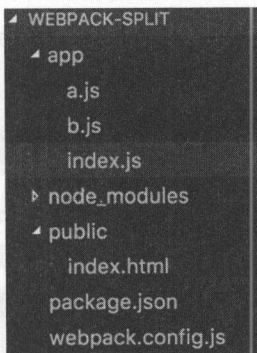


图 8.13 项目目录结构

index.js 是入口脚本文件，代码如下：

```
var root = document.getElementById('root'); // 获取根元素
function setContent(content) { // 设置根元素内容
    root.innerText = content;
}
var value = Math.floor(Math.random() * 10) % 2; // 计算出一个随机数，值为 0 或 1
if (value === 0) { // 如果随机数值为 0，加载 a.js 模块
    require.ensure(["./a"], function(require) { // 加载 a.js 模块
        var a = require("./a"); // 执行 a.js 模块
        setContent(a); // 设置内容为 a.js 模块的返回值
    });
} else { // 如果随机数值为 1，加载 b.js 模块
    require.ensure(["./b"], function(require) { // 加载 a.js 模块
        var b = require("./b"); // 执行 b.js 模块
        setContent(b); // 设置内容为 b.js 模块的返回值
    });
}
```

入口脚本文件会根据随机生成的值加载不同模块，然后将运行结果添加到页面上。

注意：require.ensure 只会加载模块，不会执行模块，所以一定要在回调函数里使用 require 执行请求到的模块。

文件 a.js 按需被文件 index.js 加载，代码如下：

```
module.exports = "I am a";
```

文件 b.js 按需被文件 index.js 加载，代码如下：

```
module.exports = "I am b";
```

注意：其他文件的作用和内容可以参考章节 8.4.1 或光盘源码，此处不再赘述。

运行应用程序，打开命令行窗口，输入如下命令：

```
npm start
```

打开浏览器，访问页面 <http://localhost:8080/index.html>，打开浏览器调试器，方便同时观察页面结果以及网络请求状态，如图 8.14 所示。

图 8.14 页面上显示的是“I am a”，说明请求的是 a.js 模块。在网络请求里面可以看到总共请求了两个资源文件。文件 bundle.js 是入口资源文件，文件 1.bundle.js 是按需加载的资源文件（由文件 a.js 打包生成）。

重复刷新页面，直到页面显示的是“I am b”，如图 8.15 所示。

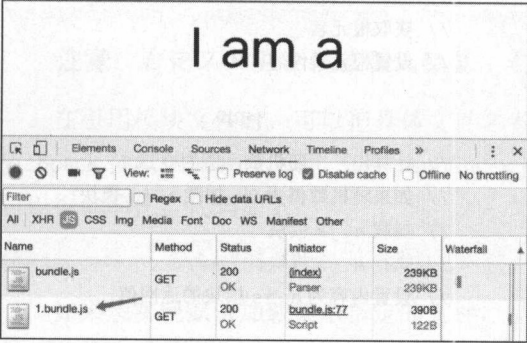


图 8.14 请求 a.js 模块的页面显示结果

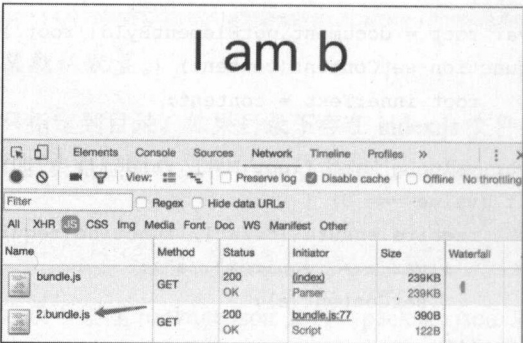


图 8.15 请求 b.js 模块的页面显示结果

图 8.15 页面上显示的是“I am b”，说明请求的是 b.js 模块。在网络请求里面可以看到总共请求了两个资源文件。文件 bundle.js 是入口资源文件，文件 2.bundle.js 是按需加载的资源文件（由文件 b.js 打包生成）。

至此，已经实现资源按需加载。这里有个问题是按需加载的资源打包之后的资源文件名是自动生成的，比如 a.js 打包之后的资源文件名是 1.bundle.js，这对调试非常不方便。Webpack 可以支持自定义异步加载的资源包文件名。修改 index.js 文件中异步调用模块的代码，相关代码如下：

```

if (value === 0) {
    require.ensure(["./a"], function(require) {
        var a = require("./a");
        setContent(a);
    }, "a");
} else {
    require.ensure(["./b"], function(require) {
        var b = require("./b");
        setContent(b);
    }, "b");
}

```

// 如果随机数值为 0，加载 a.js 模块
 // 加载 a.js 模块
 // 执行 a.js 模块
 // 设置内容为 a.js 模块的返回值
 // 这个参数设置资源的名字
 // 如果随机数值为 1，加载 b.js 模块
 // 加载 a.js 模块
 // 执行 b.js 模块
 // 设置内容为 b.js 模块的返回值
 // 这个参数设置资源的名字

其主要更改是在调用 `require.ensure` 函数的时候，多传递一个参数，这个参数代表资源的名字。修改 `webpack.config.js` 的配置，相关配置代码如下：

```

{
    output: {
        path: __dirname + "/public", // 资源输出路径
        filename: "bundle.js", // 资源文件名
        // 按需加载资源的文件名，name 在 require.ensure 中设置
        chunkFilename: "[name].js"
    }
}

```

重启应用，再次访问页面，异步加载资源的名字发生变化，如图 8.16 所示。

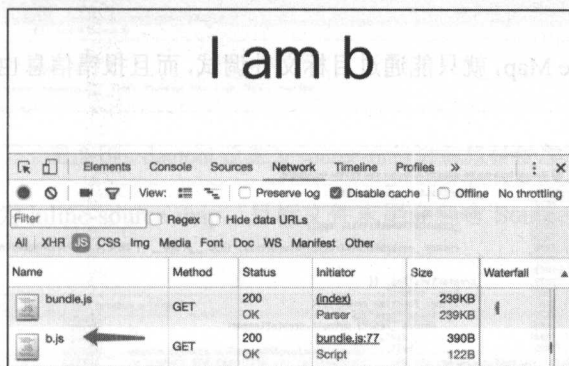


图 8.16 自定义名称的按需加载资源

8.4.6 使用 Source Map 调试代码

使用 Webpack 编译打包之后，所有源代码都经过了转译（如 ECMAScript 6 被转译成了

ECMAScript 5), 而且转译后的代码又跟 Webpack 生成的基本支持代码一起被添加到同一个文件, 因此代码调试变得非常困难。

Source Map 提供了源文件代码到目标文件代码的对应关系, 浏览器可以根据 Source Map 提供的对应关系, 支持直接调试源代码。下面通过例子说明如何使用 Webpack 来配置 Source Map, 方便调试代码。

源文件 index.js 内容代码如下:

```
import React, { Component } from 'react';    // 引入 react 包
import ReactDOM from 'react-dom';           // 引入 react-dom 包
class App extends Component {               // 定义入口组件
  render() {                                // 绘制组件
    throw "test";                           // 抛出异常, 方便查看调试结果
    return (
      <div>
        Hello World
      </div>
    )
  }
}
ReactDOM.render(                            // 绘制组件到页面
  <App />,
  document.getElementById('root')           // 根元素
);
```

如果没有配置 Source Map, 就只能通过目标文件调试, 而且报错信息也是以目标文件为标准, 如图 8.17 所示。

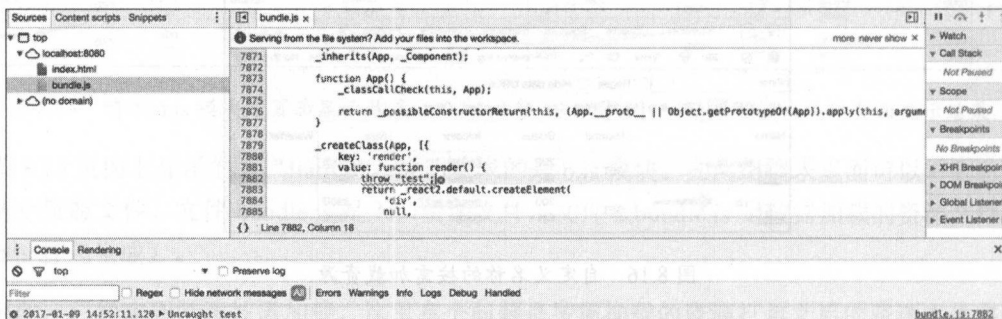


图 8.17 没有配置 Source Map 时的报错信息

Source Map 提供不同的调试机制, 在 Webpack 里使用 Source Map 时, webpack.config.js 文件

相关的配置代码如下：

```
{
  devtool: "source-map" // 配置 Source Map 调试机制
}
```

如果 dev-tool 配置为 source-map，目标文件末尾会包含 SourceMappingURL，如图 8.18 所示。

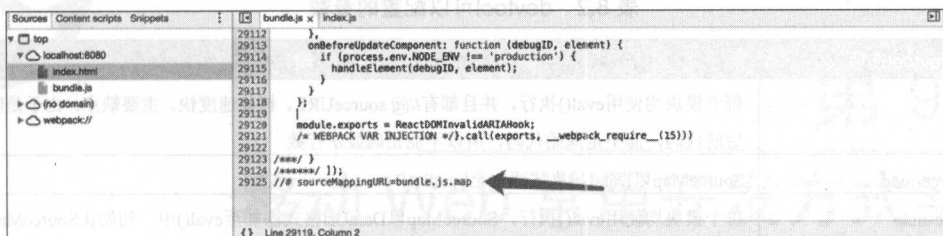


图 8.18 devtool 设置为 source-map 时的目标文件

报错信息会精确到源文件的某行某列（光标会定位到报错的列），并支持源文件的断点调试，如图 8.19 所示。

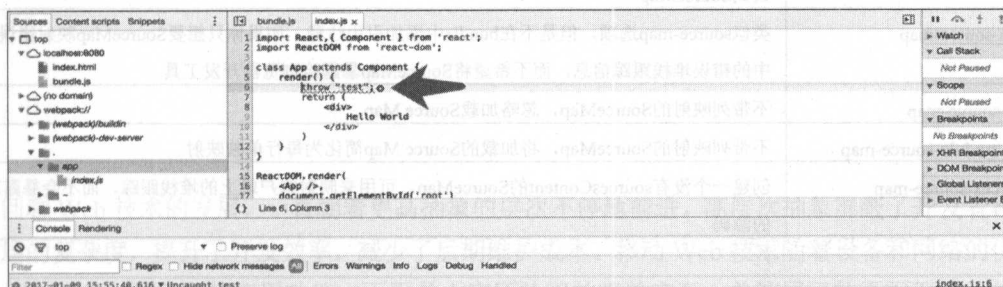


图 8.19 devtool 设置为 source-map 时的报错信息

如果 devtool 配置为 inline-source-map，目标文件末尾会包含 Source Map 的 Data URL，如图 8.20 所示。

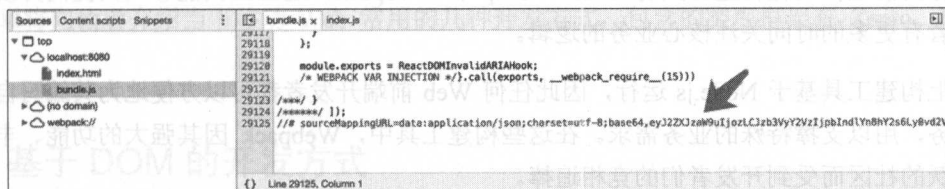


图 8.20 devtool 设置为 inline-source-map 时的目标文件

报错信息也会精确到源文件的某行某列（光标定位在报错的列），并支持源文件的断点调试。
source-map 跟 inline-source-map 提供的调试功能是一样的，只不过前者的映射信息存储在外部文件，而后者的映射信息以 Data URL 的形式存储在目标文件里。

devtool 可以配置的参数，如表 8.2 所示。

表 8.2 devtool 可以配置的参数

devtool选项	说 明
eval	每个模块均使用eval()执行，并且都有//@ sourceMappingURL。构建速度快。主要缺点：由于是映射转换后的代码，而不是原始代码，所以不能正确显示行数
inline-source-map	SourceMap以DataUrl格式添加至bundle中
eval-source-map	每个模块均使用eval()执行，SourceMap以DataUrl格式添加至eval()中。初始化SourceMap时比较慢，但重建时速度很快，并且生成实际的文件。行数能够正确映射，因为会映射到原始代码
cheap-module-eval-source-map	类似eval-source-map选项，每个模块均使用eval()执行，SourceMap以DataUrl格式添加至eval()中。“低开销”是因为其没有生成列映射，只是映射行数
source-map	以独立文件形式生成SourceMap。由于在bundle中添加了引用注释，所以开发工具知道在哪里去找到SourceMap
hidden-source-map	类似source-map选项，但是不在bundle中添加引用注释。如果你只想要SourceMap映射错误报告中的错误堆栈跟踪信息，而不希望将SourceMap暴露给浏览器开发工具
cheap-source-map	不带列映射的SourceMap，忽略加载Source Map
cheap-module-source-map	不带列映射的SourceMap，将加载的Source Map简化为每行单独映射
nosources-source-map	创建一个没有sourcesContent的SourceMap。可用来映射客户端上的堆栈跟踪，而不会暴露所有的源码

8.5 本章小结

本章主要讲解了前端工程化的相关知识。从前端的发展历程来看，工程化是必经之路。目前比较流行的前端自动化构建工具有 Grunt、Gulp 和 Webpack，这些工具能够大大提高开发效率，使开发者有更多的时间关注核心业务的逻辑。

以上构建工具基于 Node.js 运行，因此任何 Web 前端开发者都可以方便地为其编写自定义插件、任务，用以支撑特殊的业务需求。在这些构建工具中，Webpack 因其强大的功能、丰富的插件和活跃的社区而受到开发者们的竞相追捧。

本章只是简单介绍了这些构建工具的基本用法，若想更深入地了解它们，请查阅其官方文档。

9

第 9 章

移动 Web 常用开发方式实战

回看 Web 技术的发展，一直朝着更高抽象的层次不停地前进。高层次抽象屏蔽了开发者设计业务底层的复杂度，提升了开发效率，减少了后期维护成本。移动 Web 技术随着设备和网络的优化迭代，包含更多、更复杂的应用功能，不再是大家所认识的简单交互、单纯展示的页面而已。如今，移动 Web 开发者已渐渐从早期基于 DOM 的开发方式，逐渐向 MVC/MVVM 类库框架迁移，比较有代表性的框架有 React 和 Vue.js，原有的多页开发模式变为单页应用模式，Web 应用开始更需要通过团队协作的方式进行研发，这是一个必经的阶段，也是计算机发展的普遍规律。

本章将向读者介绍当下移动 Web 常用的几种开发方式，包含的类库框架有 Zepto、React 和 Vue.js。

9.1 基于 DOM 的开发方式

文档对象模型（Document Object Model，简称 DOM），是 W3C 组织推荐的处理可扩展标志

语言的标准编程接口。由于其 API 众多且在早期的浏览器中执行规范不一,常常导致一些兼容性问题,这也是缺乏经验的开发者困惑的来源。直到 jQuery 的出现,让基于 DOM 操作的开发方式变得简单,其强大和简洁的 API 以及链式调用等一大批优秀的特性被开发者推崇,这也让 jQuery 成为 Web 前端历史上最为成功的类库之一。随着移动时代的到来,移动 Web 端出现了一款与 jQuery 同样优秀的类库 Zepto,本节将围绕 Zepto 向读者介绍基于 DOM 的开发方式。

9.1.1 使用 Zepto 和前端模板开发简单备忘录

Zepto 是一款小巧的 JavaScript 类库,其大多数 API 都与 jQuery 保持一致,这让已经掌握 jQuery 的开发者使用 Zepto 不存在任何障碍,由于其核心代码压缩后大小不到 10KB,使其成为手机端一个常常被开发者使用的 JavaScript 基础类库。本节首先从 Zepto 出发,结合模板引擎 Handlebars 来完成一个简单的备忘录应用。

简单的备忘录包含一个增加待办事项的输入框及所有事项的列表,其中事项列表中的每个事项可以进行删除等操作。下面首先来完成 HTML 部分,代码如下:

```
01 <header>
02   <h1>Todos</h1>
03   <input class="new-todo" type="text">
04 </header>
05 <ul class="todos"></ul>
06 <!--单个待办事项模板-->
07 <script id="entry-template" type="text/x-handlebars-template">
08   <li class="todo">
09     <input class="toggle" type="checkbox" {{#if done}} checked {{/if}}/>
10     <label>{{title}}</label>
11     <a class="destroy">删除</a>
12   </li>
13 </script>
```

提示: 代码中对于 Handlebars 模板部分采用了客户端编译的方式,因此需要引入完整的 handlebars.js,也可以使用 Handlebars 提供的预编译工具,关于 Handlebars 的使用方法 & 语法,可以查看官方文档 <http://handlebarsjs.com>。

紧接着使用 Zepto 进行基本的 DOM 操作及事件绑定,代码如下:

```
01 var $_input = $('new-todo'),
02     $_todos = $(".todos"),
03     $_entry = $("#entry-template"),
04     compiler = Handlebars.compile($_entry.html());
```

```

05 /* 输入新的待办事项 */
06 $_input.on('keydown', function(e){
07     if (e.keyCode === 13) {                                // 回车响应
08         var data = {"title": $_input.val(), "done": false};
09         $(compiler(data)).appendTo($_todos);
10         $_input.val('');
11     }
12 });
13 /* 删除待办事项 */
14 $_todos.on('click', function(e){                            // 事件委托
15     var target = e.target || e.srcElement;
16     if(target.nodeName === 'A') {
17         $(target).parent().remove();                        // 删除当前项
18     }
19 });

```

以上的代码，首先使用了 Zepto 的选择器，将 DOM 对象转换成 Zepto 对象，并对输入框及事项列表进行事件绑定。当输入框确定输入以后，一个新的事项经过 Handlebars 的编译函数返回 HTML 代码，并将这段 HTML 代码转换成 Zepto 对象插入 DOM 树中完成页面渲染。而事项列表元素，则通过事件委托的方式，监听 Click 事件，当 Click 事件的触发对象为“删除”按钮时，将对应行从 DOM 树中删除。至此，一个简单的备忘录应用就完成了。需要说明的是，由于所有的操作均由前端通过操作 DOM 完成，后端的数据并没有发生改变，在实际的项目中，还需要通过 AJAX 或其他方式将数据同步至服务器。

9.1.2 解决原生单击事件的缺陷

2007 年，在苹果公司发布首款 iPhone 之前，由于当时的网站普遍为大屏幕设备所设计，为了应对 iPhone 这种小屏幕设备浏览桌面站点的问题，苹果的工程师引入了多项变革，其中包含“双击缩放准确定位正文主体，并将其缩放至适合的比例展现”的功能。然而，正是由于用户可以进行双击缩放的操作，在用户第一次单击页面元素的时候，浏览器并不能知道用户是想做双击缩放操作还是只是普通的单击操作。因此，iOS Safari 首先引入了 300 毫秒延迟，用来判断用户是否会再次单击，也就是说 Click 事件将会延迟 300 毫秒才得以触发。

鉴于 iPhone 的巨大成功，其他浏览器厂商纷纷效仿了 iPhone Safari 的做法，虽然从当时来看并没有什么不妥，然而在越来越注重用户体验的移动互联网时代，普通的单击都有 300 毫秒的延迟是无法被用户所接受的，加之开发者也可以根据设备尺寸对站点进行响应式适配，双击缩放的操作逐渐淡出人们的视野，浏览器厂商也意识到延迟产生的体验问题，提出了一些解决方案。

其中, 最直接的方案是添加 Meta 标签设置 Viewport 禁用页面缩放功能, 代码如下:

```
<meta name="viewport" content="user-scalable=no">
<meta name="viewport" content="initial-scale=1,maximum-scale=1">
```

Android 平台的 Chrome 浏览器率先做出了改变, 随后 Firefox 也实行了同样的约定, 在禁用缩放的页面中, 由于不存在双击缩放的可能, 直接去除了单击的延迟。这个方案看似完美, 背后却以牺牲整个页面缩放为代价, 带来的影响是对于页面上的图片和较小的文字, 想要进行缩放变得难以完成。

随后 Chrome 的团队在 Chrome 32 之后的版本中, 提出了新的方案, 代码如下:

```
<meta name="viewport" content="width=device-width">
```

在 ViewPort 设置为设备宽度的网页中, 去除了双击缩放的约定, 用户仍然可以通过双指来缩放页面上的图文内容。因为不存在双击缩放的需求, 也就去除了单击所带来的延迟。

另一个称之为指针事件 (Pointer Events) 的解决方案最早由微软提出, 指针事件是一个单独的事件模型的规范, 在这个规范中和延迟相关的实现称为 “touch-action” 的 CSS 属性, 将元素的 “touch-action” 设置为 “none” 或者 “manipulation” 可以禁用双击缩放, 从而去除单击延迟。Windows Phone 平台在 Internet Explorer 10 中率先引入了 “touch-aciton”, 而后 Chrome 以及 Firefox 也在后续的版本将其引入。

iOS 平台的 Safari 浏览器并没有立即采用其他浏览器厂商提供的方案, 而是使用了另一种方式来消除单击延迟造成的影响。在 iOS 8 版本的 Safari 浏览器中, 浏览器将会计算从 touchstart 事件到 touchend 事件所经过的时间, 并按照一个固定的临界值 (大致是 125 毫秒) 区分出两种不同的单击, 将小于临界值的单击称为 “Fast Tap”, 而将大于这个临界值的单击称为 “Slow Tap”。对于 “Fast Tap” 浏览器认为用户是打算进行双击缩放操作, 仍然会延迟 300 毫秒再执行相应 Click 事件, 而对于 “Slow Tap”, 浏览器认为用户是在进行单击操作, 会立即执行对应的 Click 事件, 而不存在延迟。

由于各个平台在实现上的不一致, 目前在解决移动端浏览器 300 毫秒延迟上比较优秀的方式是使用一款叫作 FastClick 的轻量级库, 官网地址为 <https://github.com/ftlabs/fastclick/>。FastClick 通过 MouseEvents 自定义鼠标事件并立即触发事件响应, 解决了单击 300 毫秒延迟的问题。使用该工具库非常简单, 只需将 FastClick 实例作用在 Body 元素上即可, 代码如下:

```
if ('addEventListener' in document) {
  document.addEventListener('DOMContentLoaded', function() {
    FastClick.attach(document.body);
  });
}
```

```
}, false);
```

提示：也可以使用遵循 CommonJS 或者 AMD 规范的模块加载器引入 FastClick，具体使用方法可参考官方网站。

9.1.3 为何抛弃掉 Zepto

前面已经讲述了关于单击事件 300 毫秒延迟产生的原因以及处理方式，而 Zepto 的 Touch 模块增加了一种新的“tap”事件来去除单击延迟，然而却产生了一个新的被称之为“点透”的问题。

“点透”问题的场景是这样的，当使用“tap”关闭一个弹出层时，在弹出层关闭后，事件被作用在“tap”下方的元素，如果这个元素存在 Click 事件监听，则 Click 事件会被触发，如果这个元素是链接或输入框，则会发生跳转或弹起屏幕键盘这样的浏览器默认行为。

为了搞清楚这个问题产生的原因，需要了解前面章节介绍的关于事件流的概念。由于 Zepto 是通过事件委托的方式，在 Document 文档上监听了一系列的 Touch 事件用来实现自定义“tap”事件，而在事件冒泡到 Document 之前，Touchstart、Touchend 和 Click 事件实际上都已经被触发。由于 Click 事件存在 300 毫秒延迟，当延迟结束的时候，Click 事件最终作用在了之前元素的下方元素上。虽然 Zepto 有着与 jQuery 高度相似的 API，然而在很多代码的实现上与 jQuery 并不一样，“点透”也仅仅只是 Zepto 众多问题中一个被人熟知的问题而已。

出色的类库一定会有不错的执行效率，接着不妨看看原生 JavaScript、jQuery 与 Zepto 在 DOM 操作上的性能对比，见表 9.1。

表 9.1 原生 JavaScript、jQuery 与 Zepto 在 DOM 操作上的性能对比

项 目	测 试 代 码	每秒操作次数
原生 JavaScript	<pre>var el = document.createElement('div'); el.setAttribute('id', 'elemId'); el.className = 'someClassName';</pre>	1 618 179 ± 3.51% 最快
jQuery	<pre>var el = jQuery('<div>').attr('id', 'elemId').addClass('className');</pre>	327 444 ± 1.30% 比原生慢79%
Zepto	<pre>var el = Zepto('<div>').attr('id', 'elemId').addClass('className');</pre>	99 365 ± 3.02% 比原生慢94%

从表 9.1 可以发现，Zepto 在基础的 DOM 操作上的执行效率也是远远差于其模仿对象 jQuery 的。作为一个基于 DOM 编程方式为移动端定制类库，Zepto 无论是在代码质量还是在性能等方

面都差强人意,其唯一的优点就是不到 10KB 的体积,然而代码的体积通常并不是影响整体速度的决定因素,在 jQuery 抛弃掉低版本的 Internet Explorer 浏览器以后, jQuery v2 的体积也仅仅只有 30KB 左右。

近年来,随着前端应用变得越来越复杂,为了使前端应用更加易于维护并且降低开发成本,越来越多经典的后端编程思想以及更佳优秀的设计思路正逐步应用到 JavaScript 的框架中,下面来看看目前主流的 JavaScript 框架所融入主要设计思想。

1. MVC 以及 MVVM

MVC 是模型 (Model)、视图 (View)、控制器 (Controller) 的缩写,而 MVVM 则对应着模型 (Model)、视图 (View)、视图模型 (ViewModel)。MVC 是一种软件设计典范,用一种业务逻辑、数据、界面显示分离的方法组织代码,而 MVVM 是 MVP (Model-View-Presenter) 模式演变过来的一种新型架构框架。Backbone、Knockout 以及 AngularJS 正是由后端 MVC 与 MVVM 思想而产生的前端 JavaScript 框架,而 Vue.js 与 React 则是简化了的 View 层的实现。

2. 前端路由

在单页应用中,前端路由实质上是 MVC 的 Controller 层的实现,通过路由的切换,实现与服务器的数据交换,在主流框架中,大都有前端路由的各种实现。

3. 观察者模式

观察者 (Observer) 模式是 23 种经典设计模式之一,同样是后端思想的前端应用。观察者模式定义了一种一对多的依赖关系,让多个观察者对象同时监听某一个主题对象。当这个主题对象在状态上发生变化时,会通知所有观察者对象,使其能够自动更新,这一特性,让前端框架可以在数据发生改变时,自动更新其所对应视图。

4. 模块化以及组件化

模块化与组件化的意义在于最大化的设计重用,以最少的模块、零部件,更快速地满足更多的个性化需求。提起模块化,很多有过 JavaScript 开发经验的开发者会想起 RequireJS。在早期的 JavaScript 开发中,由于 JavaScript 语言层面缺乏足够支持以及开发者之前存在个体差异,模块化以及组件化并不容易实现。目前主流 JavaScript 框架都很好地实现了模块化的编程思想,内置了模块加载器、依赖注入等模块化的功能。

5. 数据绑定与状态管理

数据绑定又分为单向数据绑定与双向数据绑定,在 AngularJS 大行其道的日子里,双向数据绑定被开发者接纳。现今流行的 React 框架则是通过父组件将状态传递给子组件,并通过 Redux 实

现了单向数据流，无论怎样，开发者不需要再去关心 DOM 操作，而只关心如何组织数据即可。

6. Virtual DOM

DOM 操作的“先天即慢”常常被认为是 Web 性能瓶颈之一。Virtual DOM 作为 React 的核心技术之一，对 DOM 进行了一层抽象，通过 JavaScript DOM 模型树（VTree）以及 DOM 模型树的差异算法（DOM Diff）提升了 DOM 操作的性能。

正是由于使用简单的 DOM 操作在构建大型前端应用的场景下愈发力不从心，无论是 Zepto 还是 jQuery 实质上只是封装了 DOM 操作的 API 类库，解决了不同浏览器带来的兼容性问题，但缺乏对前端应用整体架构设计的支持。而在主流的 JavaScript 框架中，DOM 操作作为框架的附加功能被封装在其内部实现中，基于 DOM 操作的 Zepto 或者 jQuery 仅仅是类库层面的支持，在构建大型前端应用时缺乏竞争力，Zepto 更是由于自身的缺陷，变得并不那么适用，因此，抛弃 Zepto 的时刻恐怕并不会太远了，甚至，基于 DOM 操作的开发方式，也会随着移动 Web 项目复杂度的提升，慢慢地被开发者所遗弃。

9.2 基于 React 的开发方式

React 是 Facebook 推出的一个用于构建用户界面的 JavaScript 类库，是基于组件开发模式并且实现的虚拟 DOM，极大地提高了开发效率和性能。在 GitHub 上有大量的基于 React 开发的第三方组件，丰富的组件库和活跃的社区，让 React 成为目前最流行的 JavaScript 类库之一。本节的内容将带着读者学习如何使用 React 来构建一个应用。

9.2.1 使用 JSX 语法创建 React 组件

JSX 即 JavaScript XML，在 JavaScript 语言的基础上进行扩展，支持 JavaScript 语法和类似 HTML 的标签语法的混合使用。下面是一个使用 JSX 语法的例子，代码如下：

```
ReactDOM.render(<a href="xxx">hello world!</div>, document.getElementById('root'));
```

如果不使用 JSX 语法，而使用 React 提供的 JavaScript API 实现，上述例子等同于如下代码：

```
var el = React.createElement('a', { href: 'xxx' }, 'hello world!');
ReactDOM.render(el, document.getElementById('root'));
```

使用 JSX 语法，就像调用 React.createElement 方法一样，返回一个 ReactElement 对象。其标签的名称、属性和文本都会作为参数传递给 React.createElement 方法。官方推荐的 JSX 语法，可以使大段的 HTML 标签更简洁、更清晰易读，下面开始介绍 JSX 语法的一些使用方法。

1. 标签名

JSX 语法, 支持原生的 HTML 语法。JSX 的自定义标签为了与 HTML 中原生标签区分开来, 通常使用小写开头的命名规则, 如: “<div></div>”。JSX 自定义组件则使用大写开头的驼峰式 (UpperCamelCase) 命名规则, 如: “<MyComponent></MyComponent>”。

2. 使用表达式

在 JSX 中展示动态的数据是很常见的需求, 在需要绑定数据的地方, 不管是属性还是文本, 都可以把变量或表达式放在大括号 “{}” 中插入到 JSX, 实例代码如下:

```
render() {
  var loginBtn = <button onClick={this.handleClickLogin}>登录</button>
  var isLoggedIn = this.checkLogin();
  return (
    <div>
      <h1>Hello world!</h1>
      {isLoggedIn && loginBtn}
    </div>
  )
}
```

注意: 不能在 “{}” 中直接使用 “if...else...” 语句, 这样会抛出语法错误, 可以使用三元运算符来替代。

在 JSX 中, 如果直接使用双引号为属性赋值, 赋值都会被当做字符串类型来传递, 所以如果不想传递字符类型时, 例如布尔类型或数值类型, 需要使用 “{}” 标识符来绑定数据, 实例代码如下:

<Foo active="true" />	// 按字符类型传递
<Foo active={true} />	// 按布尔类型传递
<Bar value="123" />	// 按字符类型传递
<Bar value={123} />	// 按数值类型传递

3. 延展属性

先看下面一个例子, 代码如下:

```
<MyComponent foo={1} bar={2} />
var props = { foo: 1, bar: 2 };
<MyComponent {...props} /> // 等同于第 1 行直接把属性写在 JSX 中
<MyComponent {...props} foo={3} /> // foo=3 会覆盖 props 中的 foo=1
```

上面例子中的 “{...props}” 将对象 props 的所有属性复制到自定义组件 MyComponent, 之后

设置的属性会覆盖前面的设置。延展属性解决了属性众多时依次编写的麻烦,让 JSX 结构看起来更清晰。

4. 使用注释

在 JSX 语法中使用注释就像使用 JavaScript 表达式一样,如果要在某个标签的文本区域内使用注释,可以将注释放在“{}”中,代码如下:

```
var content = (
  <Nav>
    { /* 子注释,在注释周围使用{} */ }
  <Person
    /* 多行
    注释 */
    name={window.isLoggedIn ? window.name : ''} // 行内注释
  />
</Nav>
);
```

5. JSX 中的 false

false 关键字在 JSX 中有着特殊的使用方法,实例代码如下:

```
ReactDOM.render(<div id={false} />, document.body); // 等同于 id="false"
ReactDOM.render(<input value={false} />, document.body); // 字符串"false"作为输入值
ReactDOM.render(<div>{false}</div>, document.body); // 没有子元素
```

6. HTML 实体

如果要在动态内容中显示 HTML 实体,会遇到双重转义问题,因为 React 会将所有显示的字符串转义,以便在默认情况下防止大范围的 XSS 攻击。如果要显示 HTML 实体,可以直接在“{}”中以字符串输出,实例代码如下:

```
<div>{'First &middot; Second'}</div> // 显示'First &middot; Second'
<div>{'First · Second'}</div> // 直接使用 Unicode 字符
<div>{'First \u00b7 Second'}</div> // 使用编码
<div>{'First ' + String.fromCharCode(183) + ' Second'}</div> // 使用编码
```

如果需要在 JSX 中插入原生的 HTML 片断,需要采用下面的方式,代码如下:

```
<div dangerouslySetInnerHTML={{__html: 'First &middot; Second'}} />
```

提示: XSS 全称 Cross Site Scripting,中文意思是跨站脚本攻击。攻击者通过在 Web 页面内插入恶意脚本代码,当用户访问时,嵌在页面内的代码会被执行,达到恶意攻击用户的目的。

9.2.2 在实践中掌握 React 生命周期

生命周期函数是 React 组件的重要内容,在 React 组件的不同运行时机下会被调用,某些事情只能在特定的生命周期函数中才能正确地执行,下面先来看一看 React 组件有哪些生命周期函数。

- `componentWillMount()`

在组件挂载之前立即被调用,即在初始化 `render` 之前,因此在此函数中改变 `State` 不会触发重新渲染,也不可以在此函数中进行 DOM 相关的操作。在此函数中可以访问组件的 `Props`,如果需要从远端加载数据,可以在此函数中进行操作。

- `componentDidMount()`

此函数在组件挂载之后立即被调用,在此方法中改变 `State` 会触发重新渲染,在此函数中对 DOM 进行操作是安全的。

- `componentWillReceiveProps(nextProps)`

此函数在已挂载的组件接收到新的 `Props` 时被调用,当需要响应 `Props` 的变更并对 `State` 进行修改时,可以在此方法中进行。`this.props` 可以访问旧的 `Props`,参数 `nextProps` 可以访问新的 `Props`。需要注意的一点是,此方法被调用后,`Props` 也有可能没有变更,因为父组件也会导致子组件的重新渲染,所以最好的方法是,比较 `this.props` 和 `nextProps` 中的数据是否变化,只有在发生变化的时候才去做一些响应。

- `shouldComponentUpdate(nextProps, nextState)`

此函数在组件接收到新的 `State` 或 `Props` 之后,重新渲染之前调用,默认返回 `true`。默认情况下,每次组件的 `State` 或 `Props` 变化后,都会重新渲染。如果确定某些 `State` 或 `Props` 的变化不需要重新渲染,为了提高性能,可以在此函数中进行判断,并返回 `false`,组件便不会重新渲染。

- `componentWillUpdate()`

此函数在组件重新渲染之前立即被调用,在组件第一次渲染前不会被调用,在 `shouldComponentUpdate` 返回 `false` 的情况下也不会被调用,不要在此函数中进行 `this.setState()` 操作。

- `componentDidUpdate()`

此函数在组件重新渲染之后立即被调用,在组件第一次渲染后不会被调用,在 `shouldComponentUpdate` 返回 `false` 的情况下也不会被调用。此函数是 DOM 更新后进行操作的一个好时机。

- `componentWillUnmount()`

此函数在组件被卸载和销毁前立即被调用,在此函数中可以做一些清除重置的操作,例如清

空定时器、取消网络请求和移除事件的监听等。

为了更直观地理解组件的生命周期过程，如图 9.1 所示。

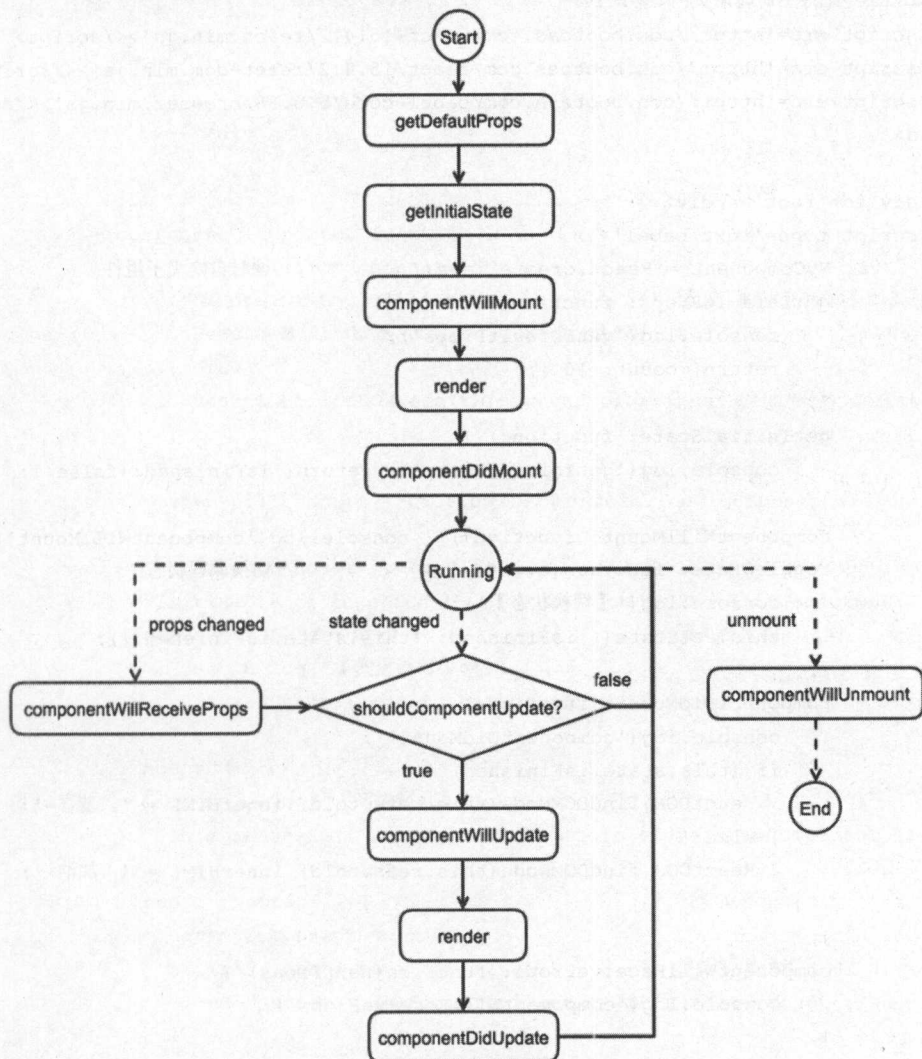


图 9.1 组件生命周期

下面通过一个实例来了解组件的生命周期过程，代码如下：

```
01 <!DOCTYPE html>
```



```

02 <html lang='en'>
03 <head>
04   <meta charset='UTF-8' />
05   <title>组件生命周期实例</title>
06   <script src='http://cdn.bootcss.com/react/15.4.2/react.min.js'></script>
07   <script src='http://cdn.bootcss.com/react/15.4.2/react-dom.min.js'></script>
08   <script src='http://cdn.bootcss.com/babel-core/5.8.38/browser.min.js'></script>
09 </head>
10 <body>
11   <div id='root'></div>
12   <script type='text/babel'>
13     var MyComponent = React.createClass({           // 课时自定义子组件
14       getDefaultProps: function() {
15         console.log('getDefaultProps');
16         return{ count: 10 };
17       },
18       getInitialState: function() {
19         console.log('getInitialState'); return{ isFinished: false };
20       },
21       componentWillMount: function() { console.log('componentWillMount'); },
22       toggleState: function(event) {               // 切换课程状态
23         console.log('【切换状态】');
24         this.setState({ isFinished: !this.state.isFinished });
25       },
26       componentDidMount: function() {
27         console.log('componentDidMount');
28         if (this.state.isFinished) {
29           ReactDOM.findDOMNode(this.refs.told).innerHTML = '，恭喜~';
30         } else {
31           ReactDOM.findDOMNode(this.refs.told).innerHTML = '，加油~';
32         }
33       },
34       componentWillReceiveProps: function(nextProps) {
35         console.log('componentWillReceiveProps');
36       },
37       shouldComponentUpdate: function(nextProps, nextState) {
38         console.log('shouldComponentUpdate'); return true;
39       },
40       componentWillUpdate: function() { console.log('componentWillUpdate'); },
41       render: function() {                          // 组件渲染逻辑
42         console.log('render');

```

```

43         return(
44             <div>
45                 本课程共{this.props.count}课时, 您
46                 <a href="javascript:void(0)" onClick={this.toggleState}>
47                     {this.state.isFinished ? '已经学完' : '还未学完'}
48                 </a>
49                 <span ref='told'></span>
50             </div>
51         )
52     },
53     componentDidMount: function() {
54         console.log('componentDidUpdate');
55         if (this.state.isFinished) {           // 根据完成状态显示不同提示
56             ReactDOM.findDOMNode(this.refs.told).innerHTML = ', 恭喜~'
57         } else {
58             ReactDOM.findDOMNode(this.refs.told).innerHTML = ', 加油~';
59         }
60     },
61     componentWillUnmount: function() { console.log('componentWillUnmount'); }
62 });
63 var MyContainer = React.createClass({           // 父组件
64     getInitialState: function () { return{ count: 10, showComponent: true }; },
65     resetCount: function() {                   // 随机设置课时时间
66         console.log('【变更课时】');
67         this.setState({ count: Math.round(Math.random() * 10) + 10 });
68     },
69     unmountComponent() {
70         console.log('【卸载组件】');
71         this.setState({ showComponent: !this.state.showComponent });
72     },
73     render: function() {                       // 组件渲染逻辑
74         var myComponent = null;
75         if (this.state.showComponent) {        // 根据状态确定是否显示子组件
76             myComponent = <MyComponent count={this.state.count} />
77         }
78         return(
79             <div>
80                 {myComponent}
81                 <button type="button" onClick={this.resetCount}>
82                     变更课时
83                 </button>

```

```

84         <button type="button" onClick={this.unmountComponent}>
85             卸载卸载
86         </button>
87     </div>
88 )
89 }
90 });
91 ReactDOM.render(<MyContainer/>, document.querySelector('#root'));
92 </script>
93 </body>
94 </html>

```

页面运行效果如图 9.2 所示。

本实例包含如下功能：

- 切换组件 MyComponent 的 State，单击“还未学完”按钮，按钮文字会变为“已经学完”。
- 更改组件 MyComponent 的 Props，单击“变更课时”按钮，课时数会随机更改。
- 卸载组件 MyComponent，单击“卸载组件”按钮，会将组件从 DOM 中移除。

按照上述的执行顺序操作，可以在控制台查看到日志信息，如图 9.3 所示。

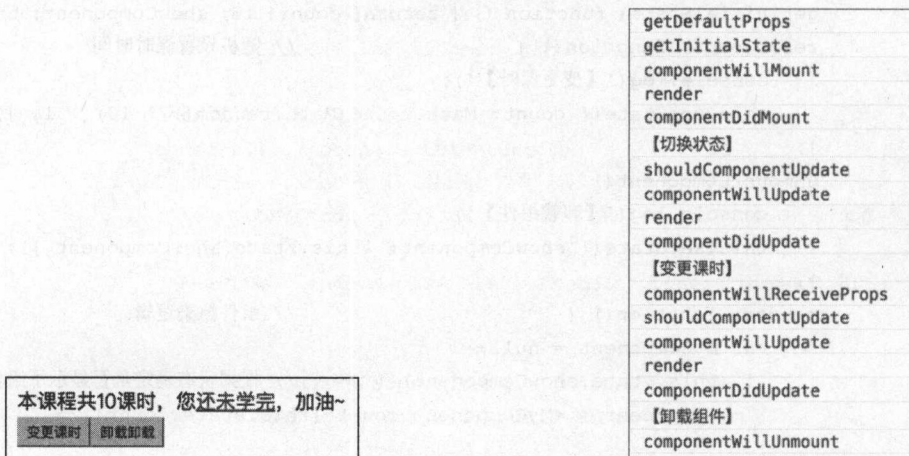


图 9.2 页面运行效果



图 9.3 控制台日志信息

图 9.3 中的日志信息，清晰地展示了组件的生命周期过程。理解组件的生命周期过程，是未来构建更复杂的 React 应用的基础，读者不妨亲自动手试一试，以便加深对 React 组件生命周期的理解。

9.2.3 实现组件间通信

一个通常的 Web 页面，由 HTML 元素一层层嵌套组合构成，React 构建 UI 的方式也进行了同样的设计，只不过是换成 React 组件而已。那么构成页面的组件之间是如何沟通、交流，数据是如何在其之间传递的？本节就来学习一下 React 组件之间的通信。React 组件间的通信包含以下两种形式。

1. 父子组件通信

- 父组件向子组件传递数据

前面的章节有提到组件实际上就是一个函数，函数所接收的参数就是组件数据的来源。参数中包括 Props 和 Children，那么很显然，父组件向子组件传递数据主要就是通过 Props 来实现。在子组件中，可以通过 `this.props` 来引用父组件传递的数据。如果父组件为子组件传递的 Props 基于其 State 或 Props 计算的值，那么只要父组件改变了相关 State 或 Props 数据，子组件的 Props 就会同步变化响应，数据就实现了从父组件到子组件的单向绑定。

- 子组件向父组件传递数据

子组件向父组件传递数据也需要借助 Props 属性，只不过并非直接传递数据，而是通过回调的方式。父组件将回调数据通过 Props 设置给子组件，子组件在触发某一动作或事件后，执行相关的回调函数，将变化的数据传递给父组件。

下面来看一个父子组件通信的例子，代码如下：

```
01 var Search = React.createClass({
02   getInitialState: function() {
03     // 设置输入框默认值
04     return { value: this.props.defaultValue };
05   },
06   handleChange: function() {
07     // 当输入内容变化时，取得最新值并更新 state
08     this.setState({ value: ReactDOM.findDOMNode(this.refs.search).value });
09   },
10   handleSearch: function() {
11     // 单击"搜索"按钮时，调用父组件的回调函数，并传递当前输入的内容
12     this.props.onSearch && this.props.onSearch(this.state.value);
13   },
14   render: function() {
15     var isDisabled = this.state.value === ''; // 设置无内容时，搜索按钮不可单击
16     return (
17       <div>
```



```

18         <input
19             type="search"
20             onChange={this.handleChange}
21             value={this.state.value}
22             ref="search"
23         />
24         <button
25             onClick={this.handleSearch}
26             disabled={isDisabled}
27         >搜索</button>
28         <button onClick={()=>this.setState({value: ''})} >清除</button>
29     </div>
30 );
31 }
32 });
33 var Page = React.createClass({
34     handleSearch: function(value) {
35         window.alert('搜索关键字: ' + value);
36     },
37     render: function() {
38         return (
39             <div>
40                 <Search defaultValue="英语六级" onSearch={this.handleSearch} />
41             </div>
42         );
43     }
44 });
45 ReactDOM.render(<Page/>, document.querySelector('#root'));

```

上面的例子中，父组件 Page 有一个子组件 Search。子组件 Search 是一个搜索框，包含一个输入框、“搜索”按钮和“清除”按钮。当用户单击“搜索”按钮后，Search 组件通知父组件处理搜索逻辑，并返回当前输入的内容。Page 组件通过设置 defaultValue 为 Search 组件传递输入框的默认值，并通过 onSearch 回调函数，接收子组件的单击“搜索”按钮通知，得到当前输入的内容。

2. 相互独立的组件通信

父子组件可以直接通过 Props 来相互通信，那么两个相互独立的组件如何通信呢？这个也要视情况而定，一般有两种选择。

- 通过共同的父级组件来通信

例如，某一个组件 A 的数据发生变化时，希望将数据传递到另一个非父子关系的组件 B 上。

组件 A 通过回调函数一层层的将数据传给父一级组件,直到传递到与组件 B 的共同父级组件 C 上。组件 C 再通过改变 Props 将数据一层层传递到子一级组件,数据最终会传递至组件 B。这种方式实现起来有点麻烦,尤其是在两个组件层级相差太多的时候,要在每一级组件上注册回调。

- 通过事件系统通信

事件系统通信需要一个全局的事件订阅和发布者 C,当组件 A 希望在某个数据变化时得到数据,可以在 C 上订阅数据变化的事件 E,当数据的来源组件 B 改变数据的时候,可以让 C 发布事件 E,并传递新的数据。这样,组件 A 在事件 E 的回调函数中就可以得到新的数据。

这种方式可以很容易地实现组件间的数据通信,而且并不限制组件之间是何种关系,但是也有缺点,即不能清晰看出数据如何流动,因为并不确定是哪个组件触发了事件的发布。

在定义事件的同时最好为事件名指定一个命名空间,避免事件名的重复而导致混乱。另外,订阅事件要在组件的 `componentDidMount` 函数中进行,并记得在组件的 `componentWillUnmount` 函数中取消事件的订阅。

提示: Flux 是 React 官方推荐的一种数据通信模式,想了解更详细的内容可以访问官方网站 <https://facebook.github.io/flux/>。

9.2.4 实现组件关注分离

在开发中,可能会经常遇到这样的场景:两个页面或区域展示的文档结构和样式是相同的,但数据源和交互不同。该情况下如何做到最大化的功能复用,思考一下不难发现,能复用的主体是展示部分,可以把这部分进行抽离,而数据和交互部分仍然分开处理。React 提供了一种模式,即 Container-Component 模式,正是解决这种需求的不错方法。

1. Container-Component 模式

该模式主要是把 React 组件分成以下两类。

- **Container:** 容器组件,主要负责数据的获取,业务相关的交互等内容。容器组件可以由其他容器组件或展示组件组合而成,将数据通过 Props 传给其他展示组件执行,通常不会直接操作 DOM 和定义样式。
- **Component:** 展示组件,只单纯负责将获取的 Props 传来的数据进行展示,可以操作 DOM 和定义样式。通常 Props 是数据的唯一来源,涉及到业务的事件和交互都通过回调通知给父级组件进行处理。展示组件对外是封闭的,外部不需要知道组件的内部细节,只需要关注组件定义的 Props。展示组件内部的状态可以通过组件的 State 进行保存。需要注意的是,组合构成展示组件的子组件只能是其他展示组件,不可以是容器组件。

下面来看一个的例子，代码如下：

```

01 var UserList = React.createClass({
02   render: function() {
03     // 遍历列表数据，返回用户列表节点数组
04     var userList = this.props.users.map(user=> (
05       <div className="user-item">
06         <img className="user-avatar" src={user.avatar} />
07         <span>{user.name}</span>
08       </div>
09     ));
10     return (
11       <ul>{userList}</ul>
12     );
13   }
14 });
15 // 约定 props
16 UserList.propTypes = {
17   users: React.PropTypes.array // 数组类型
18 };
19 var FriendListContainer = React.createClass({
20   componentWillMount: function() {
21     // 通过 fetch 获取初始数据
22     fetch('/api/friends?id=' + this.props.userId)
23       .then(res=>res.json())
24       .then(data=>{
25         this.setState({
26           friendList: data.items
27         })
28       })
29   },
30   render: function() { return (
31     <UserList users={this.state.friendList} />
32   );
33   }
34 });
35 ReactDOM.render(<FriendListContainer userId={123} />, document.querySelector('#root'));

```

提示：实例中第 22 行使用的 `fetch` 函数，是获取资源的一个接口，类似 `XMLHttpRequest`，但更加优雅，详细的 Fetch API 内容可以参考 <https://fetch.spec.whatwg.org/>。

Container-Component 模式，实现了展示和数据的关注分离。Component 组件只要根据视觉设

计的要求将内容进行展示,并通过约定 `propTypes` 告诉使用者通过 `Props` 传递哪些数据,而不需要关注组件被谁使用和数据从哪里来。单纯的 `Component` 组件可以复用在任何有同样展示需求的地方,如果业务不同,就封装多个 `Container` 组件去包裹 `Component` 组件。

合理的划分组件的层次,也是提高组件复用性的另一个好办法,不仅要关注如何复用展示内容,也要关注如果复用业务。比如,登录框组件,如果封装成带有登录逻辑的 `Container` 组件,可以用在任何需要登录的地方,而使用者不需要关注登录内部的实现逻辑。

2. 无状态函数组件 (Stateless Functional Components)

很多时候, `Component` 组件只是根据 `Props` 传递的数据进行渲染,并没有依赖生命周期函数做些复杂的状态处理或对 `DOM` 节点的操作。此时,可以考虑使用无状态函数组件。无状态函数组件是一种更纯粹的只关注渲染的组件,只有一个 `render` 函数。因不会被实例化,所以没有生命周期,也不可以使用 `this.refs` 和 `this.state`。可以将其看成一个纯函数,接收 `Props` 作为参数,返回一个组件节点片断。下面是一个无状态函数组件的例子,代码如下:

```
var Header = props => {
  return (
    <div className="header">
      <a className="btn btn-back" onClick={props.onBack}></a>
      <span>{props.title}</span>
      <a className="btn btn-share" onClick={props.onShare}></a>
    </div>
  )
}
// 使用方法: <Header title="标题", onBack={handleBack} onShare={handleShare}/>
```

例子中的 `Header` 组件,并不是由 `React.createClass` 创建,只是一个接收 `Props` 参数的普通函数,使用方法和普通的组件相同。

3. 高阶组件 (Higher Order Components)

“高阶函数”是指将函数作为参数或返回值的函数。开发者可以使用高阶函数实现对一个函数的包装和扩展。`React` 组件本身即是函数,具备高阶函数功能,即 `React` 高阶组件。高阶组件的形式,代码如下:

```
React.Component => React.Component
```

下面的例子,使用了高阶组件实现先前实例中 `FriendListContainer` 组件,代码如下:

```
01 var FriendListEnhance = Component => React.createClass({
02   componentWillMount: function() {
```



```

03     fetch('/api/friends?id=' + this.props.userid)
04     .then(res=>res.json())
05     .then(data=>{
06         this.setState({
07             users: data.items
08         })
09     })
10 },
11 render: function() {
12     return <Component {...this.props} {...this.state} />
13 }
14 })
15 // 使用高阶组件, 为 UserList 组件绑定数据
16 var FriendList = FriendListEnhance(UserList)
17 ReactDOM.render(<FriendList userId={123} />, document.querySelector('#root'));

```

例子中, FriendListEnhance 为高阶组件, 接收一个组件 Component 作为参数, 调用好友列表相关的接口后, 将数据更新回组件。FriendList 是使用高阶组件 FriendListEnhance 传入 UserList 绑定数据后新生成的组件。类似这样展示好友列表的需求, 为了实现数据和展示的关注点分离, 通过抽象和拆分为下面的两个部分:

- 利用高阶组件 FriendListEnhance 抽象分离数据处理。
- 使用展示组件 UserList 抽象分离视图展示。

高阶组件同样可以被其他高阶组件扩展, 并且不限定扩展次数, 代码如下:

```
FriendListEnhance(OrderEnhance(FilterEnhance(UserList)))
```

使用 OrderEnhance 组件对列表数据进行排序, 使用 FilterEnhance 组件对列表数据进行过滤, 并且这两个高阶组件可以使用在任何有同样需求的组件上。

在使用 React 开发复杂业务时, 对需求的抽象分解也是非常重要的。在实际开发中灵活地运用本节介绍的几种方法, 对提高代码的维护性、复用性、扩展性会起到很大的帮助。

9.2.5 实战演练: 运用组件化方式开发一个备忘录

本节将综合前面几节学习的内容, 使用 React 组件化的方式来开发一个备忘录应用。备忘录主要包含以下几个功能点:

- 支持添加新的备忘。
- 可切换备忘的“待做”和“已完成”状态, 可设置颜色, 可删除。

- 可根据颜色、时间对列表进行排序。
- 可分别查看“全部”、“待做”及“已完成”状态的备忘。
- 支持本地数据储存 LocalStorage。

先看一下最终的运行效果，如图 9.4 所示。



图 9.4 备忘录运行效果

在开始编写代码之前，先根据页面的结构来规划将要实现的业务组件，包含以下几个方面。

- **TodoApp**: 整个应用的根组件，负责储存及处理应用的数据。
- **Header**: 展示组件，表示应用的头部，包含应用的标题、新建备忘的内容输入框、时间输入框、对备忘列表进行排序和过滤的操作栏。
- **TodoList**: 展示组件，表示应用的列表部分，包含列表标题和列表内容。
- **Memo**: 展示组件，表示单个备忘事项，可根据数据展示相应的备忘状态。由一个改变备忘状态的复选框、改变备忘颜色的颜色选择器、备忘内容、备忘时间及删除备忘按钮组成。删除按钮默认为隐藏，在鼠标悬停在单个备忘事项时显示。复选框未选中时，组件为“待做”状态；被选中时，为“已完成”状态。

Memo 组件代码如下：

```
01 /** 备忘事项组件 */
02 const Memo = props => {
```

```

03     let memo = props.memo;
04     // 根据备忘的状态设置样式
05     let classNames = 'todo-item' + (memo.done ? ' done' : '');
06     return (
07       <div className={classNames}>
08         <input type="checkbox" checked={memo.done}
09           onChange={()=>props.onToggleState(memo)} />
10         <input className="color" type="color" value={memo.color}
11           onChange={(e)=>props.onChangeColor(memo, e.target.value)} />
12         <span className="text">{memo.text}</span>
13         <span className="pull-right del"
14           onClick={()=>props.onDelete(memo)}>X</span>
15         <a className="pull-right">(new Date(memo.time).toLocaleString())</a>
16       </div>
17     );
18 }

```

提示: 在上面的例子及本节之后的代码书写上, 将会部分使用 ECMAScript 6 的语法, 有兴趣的读者可以自行了解相关的内容, 参考网址为 <http://es6-features.org/>。

Memo 组件所接收的 Props 数据结构, 代码如下:

```

{
  color: '#ffffff',           // 组件颜色
  text: 'text',              // 备忘内容
  time: '2017/2/3 11:20',    // 备忘时间
  done: false                // 备忘状态, 是否已完成
}

```

TodoList 组件代码如下:

```

01 /** 备忘列表组件 */
02 const TodoList = props => {
03   // 遍历 props 中的列表数据, 生成备忘组件列表
04   let todolist = props.items.map(memo=>{
05     <Memo memo={memo} onToggleState={props.onToggleState}
06       onChangeColor={props.onChangeColor} onDelete={props.onDelete} />
07   });
08   return (
09     <div className="todo-list-wrapper">
10       <div>{`${props.title} (${props.items.length})`}</div>
11       <div className="todo-list">{todolist}</div>
12     </div>

```

```

13 );
14 }

```

TodoList 只是一个展示组件，由父组件接收列表标题和列表数据遍历展现，没有直接处理 Memo 组件的相关回调，而是直接调用父组件设置的相应回调函数处理。

Header 组件 render 函数返回的的组件结构，代码如下：

```

01 <div className="header">
02   <h3 className="title">备忘录</h3>
03   <button type="button" disabled={!this.state.value}
04     onClick={this.handleAdd.bind(this)}>添加</button>
05   <div className="input-wrapper">
06     <input value={this.state.value} placeholder="备忘内容"
07       onChange={this.handleChange.bind(this)} />
08     <input className="date" type="date" value={this.state.date}
09       onChange={this.handleChange.bind(this)} />
10     <input className="time" type="time" value={this.state.time}
11       onChange={this.handleChange.bind(this)} />
12   </div>
13   <div className="bar">
14     <strong>排序</strong>
15     <span onClick={()=>this.props.onChangeOrder('color')}>颜色</span>
16     <span onClick={()=>this.props.onChangeOrder('time')}>时间</span>
17     <strong style={{ marginLeft: 20 /* 行内样式 */ }}>过滤</strong>
18     <span onClick={()=>this.props.onChangeFilter('全部')}>全部</span>
19     <span onClick={()=>this.props.onChangeFilter('待做')}>待做</span>
20     <span onClick={()=>this.props.onChangeFilter('已完成')}>已完成</span>
21   </div>
22 </div>

```

当用户单击排序或过滤选项按钮时，Header 组件直接调用通过 Props 设置的回调函数来通知父组件做相关的处理。

TodoApp 组件代码如下：

```

01 /** 备忘录组件 */
02 class TodoApp extends React.Component {
03   constructor() {
04     super();
05     // 设置初始状态
06     this.state = loadData() || {
07       todos: [], // 储存备忘列表

```



```

08         filter: '全部',      // 储存当前的过滤字段
09         order: 'time' // 储存当前的排序字段
10     };
11 }
12 }

```

在构造函数中, 会尝试通过 `loadData` 函数从 `LocalStorage` 中读取储存数据, 如果不存在, 则使用默认数据。

给 `TodoApp` 增加 `render` 函数, 代码如下:

```

01 render() {
02     let filter = this.state.filter;
03     // 先对列表进行过滤, 再进行排序
04     let items = sortByProp(this.state.todos.filter(memo=>{
05         return filter === '全部' ? true : filter === '已完成' ? memo.done : !memo.done;
06     })), this.state.order);
07     return (
08         <div className="todo-app">
09             <Header onClickAdd={this.handleClickAdd.bind(this)}
10                 onChangeOrder={this.handleChangeOrder.bind(this)}
11                 onChangeFilter={this.handleChangeFilter.bind(this)} />
12             <TodoList title={filter} items={items}
13                 onToggleState={this.handleToggleState.bind(this)}
14                 onChangeColor={this.handleChangeColor}
15                 onDelete={this.handleDelete.bind(this)} />
16         </div>
17     );
18 }

```

为 `TodoApp` 添加相关的处理函数, 代码如下:

```

01 /* 添加新备忘的处理函数, 向 todos 列表中插入新数据 */
02 handleAdd(text, date, time) {
03     // setState 是异步的, 在设置 state 完成后存储当前的 state
04     this.setState({
05         todos: this.state.todos.concat([
06             {
07                 color: '#ffffff',
08                 text,
09                 time: date + ' ' + time,
10                 done: false
11             }
12         ])
13     }, ()=>saveData(this.state));

```

```

12 }
13 /* 切换备忘状态的处理函数 */
14 handleToggleState(memo) {
15     memo.done = !memo.done;
16     this.setState({}, ()=>saveData(this.state))
17 }
18 /* 切换备忘颜色的处理函数 */
19 handleChangeColor(memo, color) {
20     memo.color = color;
21     this.setState({}, ()=>saveData(this.state))
22 }
23 /* 删除备忘的处理函数，将备忘从 todos 中移除 */
24 handleDelete(memo) {
25     let todos = this.state.todos;
26     todos.splice(todos.indexOf(memo), 1);
27     this.setState({ todos: todos }, ()=>saveData(this.state));
28 }
29 /* 根据指定字段对列表进行排序 */
30 handleChangeOrder(field) {
31     this.setState({ order: field }, ()=>saveData(this.state));
32 }
33 /** 根据备忘状态对列表进行过滤 */
34 handleChangeFilter(filter) { this.setState({ filter }, ()=>saveData(this.state)); }

```

其他功能函数，包含数据加载、存储和数据排序，代码如下：

```

01 function loadData() { // 读取存储的列表数据
02     try {
03         return JSON.parse(window.localStorage.getItem('todos')); // 读取数据
04     } catch(e) {
05         // 浏览器不支持 localStorage
06         return null
07     }
08 }
09 function saveData(data) { // 将数据存储至 localStorage 中
10     try {
11         window.localStorage.setItem('todos', JSON.stringify(data)); // 储存数据
12     } catch(e) {
13         // 浏览器不支持 localStorage
14     }
15 }
16 /** 根据数据元素的某个属性对数据进行排序 */

```

```

17 function sortByProp(arr, prop, reverse = false) {
18     return arr.sort((a, b) => reverse ? a[prop] > b[prop] : a[prop] < b[prop]);
19 }

```

最后, 把 `TodoApp` 组件渲染到 DOM 中, 代码如下:

```
ReactDOM.render(<TodoApp/>, document.querySelector('#root'));
```

上述例子中, 对于最基本的展示组件 `Memo` 和 `TodoList`, 使用无状态组件足以实现。把应用的数据都储存到根组件 `TodoApp` 中, 这样更易于管理, 也方便存储至离线数据 `LocalStorage` 中。备忘录是个很经典的应用, 功能虽然简单, 但可以通过本实例掌握前面所介绍的知识点, 读者也可以尝试增加更多的功能, 完善这个应用。

9.2.6 如何管理应用的状态

所谓应用的状态, 即根据数据的变化所呈现出来的不同的表现。传统 Web 页面, 应用的状态由后端服务进行维护, 页面需要通过频繁的刷新来响应状态的改变。而如今的 Web 页面, 交互变得越来越复杂, 单页应用变得越来越普遍, 经常通过异步请求让页面局部刷新的方式来获得更好的用户体验。这种变化也使得对应用状态的维护从后端转移到前端。如何管理应用的状态, 对前端来说也是一个新的难点。本节将会介绍在 `React` 组件化的前端框架下维护应用状态的几种方法。

1. 使用 Store 将数据与视图分离

上一节介绍的备忘录例子, 数据直接储存在根组件的 `State` 中, 子组件获得数据需要经过父组件一层层的传递。但对于状态复杂的应用, 数据统一储存在根 `State` 中, 使用起来既麻烦, 也不利于维护。所以, 更好的办法是, 将数据交由全局的某个对象来专门维护, 实现数据与视图关注分离。这个专门用来存储数据的对象, 就是 `Store`, 主要有以下三种功能:

- 数据存储, 存储在 `Store` 中的数据通常是私有的, 外部不可以直接修改。
- 对外提供一些对数据增加、删除、更改、查询的接口。
- 支持数据变化的通知, 方便视图及时响应数据的变化。

通过 `Store` 对象, 视图组件不再需要关心数据的来源和对数据的维护, 只需要调用 `Store` 相应的接口来取得和更新数据即可。组件间需要复用数据也不再通过层层传递, 可交由 `Store` 来专门维护, 并且可对数据做一些共通处理, 如过滤、转换、缓存等。同时, 数据的修改只能通过 `Store` 提供的接口进行, 也保护数据不会被随意变更, 方便记录和追溯数据的变化。当然, 也不是所有的数据都要存储在 `Store` 中, 一些组件的内部状态直接交由组件 `State` 维护会更清晰。

提示：对于展示组件，不要直接引用 Store，还是需要父组件通过 Props 传递。容器组件可以直接引用 Store，但要注意不要直接在 render 函数中引用，数据从 Store 中获取后先同步至组件 State 中，除非数据确定是不会变化的。

2. Flux

Flux 由 Facebook 推出，用于构建客户端 Web 应用架构。Flux 通过利用单向数据流来辅助 React 组合视图组件。Flux 更多的是一种模式，而不是一个正式的模架，有很多基于 Flux 模式的第三方框架实现，如 Reflux、Redux 等。

Flux 包括以下四个核心概念。

- **Action（动作）：**行为或动作，一个包含了动作类型的标识和新数据的简单对象，通常源自用户与视图的交互，也可以来源于定时任务或是服务器响应等。
- **Dispatcher（派发器）：**负责将 Action 派发给 Store 进行处理。
- **Store（存储）：**类似前面介绍的 Store，负责存储数据，根据 Action 对数据进行处理，并支持对数据变化事件的订阅和发布。
- **View（视图）：**在 React 中即组件，为了实现组件的关注分离，通常是 Container 组件。

Flux 的单向数据流，如图 9.5 所示。

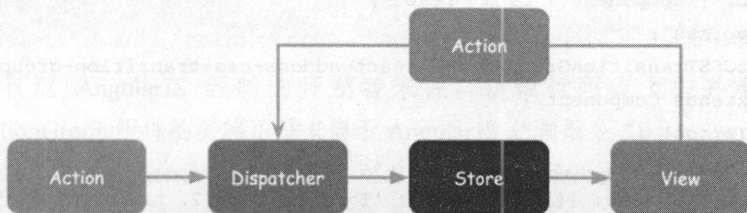


图 9.5 Flux 单向数据流示意图

所有的数据流动都要通过作为中心集线器的 Dispatcher，Dispatcher 将 Action 派发给 Store 进行处理。Store 会响应每个与保存状态有关的 Action，对指定的数据进行增加、删除、更改等操作，再将数据变更通知给相关的订阅者。View 响应从 Store 订阅的数据变更事件，将新数据通过 setState 方法同步至组件 State，触发自身和所有后代的重新渲染。

很多开发者都会觉得双向数据绑定的使用更加便利。但需要注意到的是，当应用越来越庞大时，数据变得更不可预测，应用的各个节点都有可能触发数据的修改。在 Flux 的单向数据流中，所有对数据操作的行为都要经过 Dispatcher，行为变得可被预测，数据的流动也更加清晰。

提示：目前比较热门的 Flux 实现是 Redux，读者可以访问 Redux 官网 <http://redux.js.org/> 了解更多的内容。

9.2.7 添加动画效果

在前端领域，关于动画效果的实现，常用的有两种方式，一种是使用 CSS3 的 Transition 或 Animation 功能，另一种则是通过 JavaScript 在时间变化过程中改变 DOM 元素的样式属性。

提示：HTML 5 增加的 Canvas 和 SVG 标签也可以用来实现动画效果，Canvas 动画的本质是通过 JavaScript 不断重绘画布，而 SVG 除了可以使用类似操作 DOM 的方式实现动画外，还可以用同步多媒体集成语言（SMIL）来描述动画，关于 SMIL 可以参考规范文档 <http://www.w3.org/TR/REC-smil/>。

需要思考的是 React 改变了基于 DOM 操作的编程方式。由于 React 组件生命周期的存在，使用 JavaScript 直接操作 DOM 元素来完成动画效果，在 React 项目中并不是一种很好的选择。为了实现动画效果，React 官方提供了两种组件：ReactTransitionGroup 和 ReactCSSTransitionGroup。ReactTransitionGroup 是底层组件，而 ReactCSSTransitionGroup 组件则是基于 ReactTransitionGroup，并提供了执行 CSS3 Transition 和 Animation 动画的一种简单的方式。首先看一个例子，代码如下：

```
01 import React, { Component } from 'react';
02 import './App.css';
03 import ReactCSSTransitionGroup from 'react-addons-css-transition-group';
04 class App extends Component {
05   constructor(props) {
06     super(props);
07     this.state = {items: [{id: 1, title: 'Todo1'}, {id: 2, title: 'Todo2'}]}; // 默认数据
08     this.addItem = this.addItem.bind(this);
09   }
10   addItem() {
11     let key = this.state.items[this.state.items.length - 1].id + 1;
12     const newItems = this.state.items.concat([{id: key, title: 'Todo' + key}]); // 添加项
13     this.setState({items: newItems}); // 同步 state
14   }
15   removeItem(i) {
16     const newItems = this.state.items.slice();
17     newItems.splice(i, 1); // 删除项
18     this.setState({items: newItems}); // 同步 state
19   }
20   render() {
```

```

21  const items = this.state.items.map((item, i) => {
22    <li key={item.id} onClick={() => this.removeItem(i)}>
23      {item.title}
24    </li>
25  });
26  return (
27    <div className="App">
28      <h2>React CSS 动画实例</h2>
29      <button onClick={this.addItem}>增加新项</button>
30      <ul>
31        <ReactCSSTransitionGroup
32          transitionName="react-animaition"           // 关联 CSS 类
33          transitionAppear={true}                     // 是否初始挂载动画
34          transitionAppearTimeout={500}                // 动画出现时长
35          transitionEnterTimeout={500}                 // 动画进入时长
36          transitionLeaveTimeout={500}>                // 动画离开时长
37          {items}
38        </ReactCSSTransitionGroup>
39      </ul>
40    </div>
41  );
42 }
43 }

```

对于曾经有过 AngularJS 经验的开发者来说，这段代码看上去并不会那么陌生，ReactCSSTransitionGroup 组件的灵感正是来自于 AngularJS 动画指令“ng-animate”。当子组件初始化挂载到 ReactCSSTransitionGroup 中，或者对 ReactCSSTransitionGroup 做添加项、移除项操作时，特殊的 CSS 样式类将被添加到子组件。继续上述实例，为了让 ReactCSSTransitionGroup 内的子组件产生动画效果，添加对应 CSS，代码如下：

```

01  .react-animaition-appear {
02    opacity: 0.01;
03  }
04  .react-animaition-appear.react-animaition-appear-active {
05    opacity: 1;
06    transition: opacity 500ms ease-in;
07  }
08  .react-animaition-enter {
09    opacity: 0.01;
10  }
11  .react-animaition-enter.react-animaition-enter-active {

```

```

12     opacity: 1;
13     transition: opacity 500ms ease-in;
14 }
15 .react-animaition-leave {
16     opacity: 1;
17 }
18 .react-animaition-leave.react-animaition-leave-active {
19     opacity: 0.01;
20     transition: opacity 500ms ease-in;
21 }

```

接着详细介绍 `ReactCSSTransitionGroup` 组件的参数配置。

- **transitionName**: 附加在子组件上的 CSS 样式名均基于 `transitionName` 生成, 当以字符串形式提供该参数时, `transitionName` 会作为附加样式名的前缀生成对应样式名, 也可以使用对象字面量的方式, 固定所有或者部分附加样式名, 实例代码如下:

```
"transitionName={{enter: 'enter', enterActive: 'enterActive'}}"。
```

- **transitionAppear**: 定义是否在子组件初始化挂载阶段加入附加样式, 默认值是 `false`。
- **transitionEnter**: 定义是否在新项目进入 `ReactCSSTransitionGroup` 组件时加入附加样式, 由于 `transitionEnter` 以及 `transitionLeave` 的默认值都是 `true`, 实际在上例中已经省略。
- **transitionLeave**: 定义是否在项目被移除 `ReactCSSTransitionGroup` 组件时加入附加样式。
- **transitionAppearTimeout**: 定义子组件初始化挂载阶段加入附加样式后, 到从子组件上移除这个附加样式所经过的时间。
- **transitionEnterTimeout**: 定义对添加到 `ReactCSSTransitionGroup` 内的子组件加入附加样式后, 到从这个子组件上移除这个附加样式所经过的时间。
- **transitionLeaveTimeout**: 定义将要移除 `ReactCSSTransitionGroup` 内的子组件加入附加样式后, 到从这个子组件上移除这个附加样式所经过的时间, 并且随后该子组件从 DOM 中被移除。

前文已经说明 `ReactCSSTransitionGroup` 基于 `ReactTransitionGroup` 实现, 作为底层组件, `ReactTransitionGroup` 提供特殊生命周期 hooks 函数。`ReactCSSTransitionGroup` 正是利用这些函数, 对子组件各阶段附加了对应的 CSS 样式类。

由 `ReactTransitionGroup` 提供的这些特殊生命周期函数如下。

- **componentWillAppear(callback)**: 组件被初始化挂载到 `ReactTransitionGroup` 内的 `componentDidMount` 阶段被调用, 阻止其他动画发生直到其回调函数被执行。

- `componentDidAppear()`: `componentWillAppear` 回调函数被执行后调用。
- `componentWillEnter(callback)`: 组件添加到 `ReactTransitionGroup` 内的 `componentDidMount` 阶段被调用, 阻止其他动画发生直到其回调函数被执行。
- `componentDidEnter()`: `componentWillEnter` 回调函数被执行后调用。
- `componentWillLeave(callback)`: 组件从 `ReactTransitionGroup` 内移除时调用, 尽管子组件被移除, `ReactTransitionGroup` 仍然在 DOM 保持直到回调被执行。
- `componentDidLeave()`: 组件 `componentWillUnmount` 阶段调用。

接下来可以利用 `ReactTransitionGroup` 提供的这些特殊生命周期函数, 自己定制动画效果, 也可以引入第三方动画库, 如 `Velocity.js` 来实现动画效果, 具体可以参考 React 官方文档。

提示: React 官方提供的 `ReactTransitionGroup` 只是针对基本的动画需求提出的一种解决方案, 目前还有一些高度定制化的动画组件库, 如 <https://github.com/chenglou/react-motion>, 读者可以自行参考使用。

9.2.8 提高 React 组件性能

浏览器使用渲染引擎解析 HTML 和 CSS 文件, 根据 HTML 元素创建 DOM 树, 同时对 DOM 树附加 Style 规则生成相应的 Render 树, 最终通过“paint”方法绘制出页面。通常认为频繁的 DOM 操作会导致浏览器重绘 (repaint) 以及重排 (reflow), 从而引发性能问题, 尤其对缺乏经验的开发者来说, 对优化 DOM 操作缺乏了解, 更容易带来一些不易发现的性能损耗。

React 引入了 Virtual DOM 来解决频繁的 DOM 操作带来的性能问题, 但这并不代表 React 在性能优化上已经没有空间了。`shouldComponentUpdate` 是 React 提供的一个生命周期函数, 在前面的小节中已有过介绍, 由于 `shouldComponentUpdate` 的默认返回值为 `true`, 也就是说任何 Props 或 State 的更改, 都会触发 React 组件的重新渲染过程, 代码如下:

```
shouldComponentUpdate: function(nextProps, nextState) {
  return true;
}
```

React 组件的重新渲染过程中, 会创建新的 Virtual DOM 与旧的 Virtual DOM 进行比较。在结构比较复杂的情况下, Diff (差异比较) 过程会花费较多时间, 当 Diff 的结果表示 Virtual DOM 发生变化时, React 会更新界面的 DOM 元素, 完成界面同步。然而并不是所有的 Props 或 State 变化都应该被执行上述过程, 在对组件传入相同的 Props 或 State 时, 避免不必要的重新渲染过程, 是优化 React 组件性能的重要手段。比如, 当 `shouldComponentUpdate` 返回 `false` 时, React 会跳过组件重新渲染的过程, 代码如下:


```

shouldComponentUpdate(nextProps, nextState) {
  if (this.props.color !== nextProps.color) {           // 比较 props
    return true;
  }
  if (this.state.count !== nextState.count) {           // 比较 state
    return true;
  }
  return false;
}

```

在以上代码中,实质是针对简单结构的 Props 或 State 进行了浅比较。在早期的 React 版本中,提供了 **PureRenderMixin** 来便捷地完成浅比较的过程,代码如下:

```

import PureRenderMixin from 'react-addons-pure-render-mixin';
class YourComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }
}

```

而在现行版本 React v15.3+ 中,推荐使用 **PureComponent**。**React.PureComponent** 与 **React.Component** 类似,只是在其 **shouldComponentUpdate** 生命周期函数中,实现了 Props 与 State 的浅比较。如果判断 React 组件的 **render()** 方法渲染出相同结果时,使用 **PureComponent** 会有更好的性能表现,代码如下:

```

class YourComponent extends React.PureComponent { }

```

在使用 **React.PureComponent** 进行性能优化时,需要注意的是由于数据是可变的,对可变数据进行操作时,可能存在副作用,看下这个例子,代码如下:

```

01 class Item extends React.PureComponent {
02   render() {
03     return <div>{this.props.items.join(',')}</div>;
04   }
05 }
06 class App extends React.Component {
07   constructor(props) {
08     super(props);
09     this.state = {
10       items: ['item']
11     };

```

```

12     this.handleClick = this.handleClick.bind(this);
13   }
14   handleClick() {
15     const items = this.state.items;
16     items.push('newItem');           // 引用类型的 items 被添加了新数据
17     this.setState({items});
18   }
19   render() {
20     return (
21       <div>
22         <button onClick={this.handleClick} />
23         <Item items={this.state.items} />
24       </div>
25     );
26   }
27 }

```

上述代码中，由于 `state.items` 是一个引用类型的数组对象，在执行数组的 `push` 操作时，原数组实际上已经发生了改变。而子组件 `Item` 在其 `shouldComponentUpdate` 生命周期函数中，`this.props` 以及 `nextProps` 都是对原数组的引用。在进行比较时，由于 `this.props` 和 `nextProps` 是相等的，React 认为 `Props` 并没有发生变化，所以 `shouldComponentUpdate` 返回为 `false`，也就不会执行组件的重新渲染以及 `DOM` 元素的更新。要解决这个问题，简单的方式是对原数据进行复制，这样 `this.props` 和 `nextProps` 会指向不同的引用，代码如下：

```

handleClick() {
  this.setState(prevState => ({
    items: prevState.items.concat(['newItem']) // concat 产生新数组对象
  }));
}

```

也可以使用 `Object.assign` 实现相似的效果。然而，对复杂的数据结构进行复制以及后续的比较都会带来性能损耗，更优雅的解决方案是使用不可变的数据结构。由 Facebook 工程师开发的 `Immutable.js` 正是这样一个工具。`Immutable.js` 通过结构共享提供不可变持久化集合类型，诸如 `List`、`Stack`、`Map`、`Set` 等，一旦创建，集合就不会改变。现在，只需要引入 `Immutable.js` 并使用不可变的数据结构定义之前的数据 `items`，就可以更加高效地实现与之前使用可变数据一样的效果了，代码如下：

```

01 import Immutable from 'immutable';           // 引入 immutable.js
02 class App extends React.Component {
03   constructor(props) {

```

```

04     super(props);
05     this.state = {
06         items: Immutable.List(['item'])           // 不可变的 items
07     };
08     this.handleClick = this.handleClick.bind(this);
09     }
10     handleClick() {
11         const items = this.state.items.push('new item');    // 对 items 增加新项
12         this.setState({items});
13     }
14 }

```

使用 `Immutable.js` 可以给 `React` 应用带来极大的性能提升,但是由于代码侵入性较强,原则上对于过往项目的改造,是否使用 `Immutable.js` 还需要根据项目情况进行合理的评估。最后需要说明的是, `Immutable.js` 提供的部分数据结构类型同 `ECMAScript 6` 提供的数据结构类型使用了相同的命名,需要注意区分,以免产生混淆。

9.3 基于 Vue.js 的开发方式

`Vue.js` 异军突起,已经成为了市面上最流行的前端框架之一。尤其在国內,很多团队选择 `Vue.js` 作为前端框架,可能有几方面原因,一方面在于作者是华人且在早期就有同步的中文文档,另一方面 `Vue.js` 的入门成本较低,没有对于新人晦涩的 `JSX` 语法或是复杂的依赖注入,组件化加上基于模板的双向绑定易于上手。在其 2.0 版本提出了渐进式框架和自底向上增量开发的概念,对于不同的需求可以选择不同程度的模块集成,很好地控制了项目的复杂度和包文件的体积大小,十分值得读者学习和使用。

9.3.1 实战演练: 开发一个简单的备忘录应用 (Vue.js 2.0)

使用 `NPM` 进行安装 `Vue.js` 开发环境, 命令如下:

```
npm install vue
```

然后, 在页面引入 `JavaScript`, 代码如下:

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
```

提示: 因为 `Vue.js` 使用了 `ECMAScript 5` 的特性, 所以不支持 `Internet Explorer 8` 和更低版本的浏览器。

首先介绍 Vue.js 框架特性，其编写方式采用模板、脚本和样式三个部分的传统前端模式，并且可以使用开发者熟悉的预编译语言，需要注意的是打包环境通过相应的加载器进行处理。

Vue.js 同样提供了几乎被所有新框架采用的组件系统，同时基于 ECMAScript 5 中 `Object.defineProperty` 将 Data 属性转换成 Getter 和 Setter 实现数据双向绑定。Vue.js 的响应式原理，如图 9.6 所示。

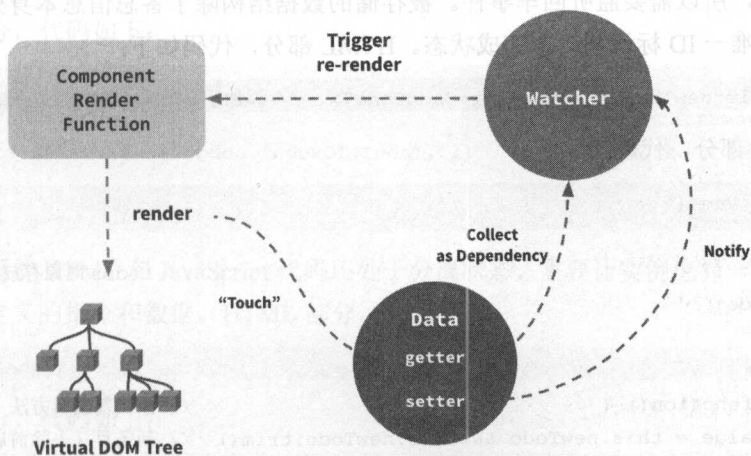


图 9.6 Vue.js 的响应式原理

在本章开头提到 Vue.js 是一个易上手的前端框架，现在通过一个备忘录来展示其实现方式。一个简单的备忘录应用页面可以分成两个部分，一个输入框用来记录输入的信息，另一个列表用来展示备忘录条目，并且可以选择完成或者删除。

(1) 制作输入框部分，最简单的输入框只需要一个 `Input` 标签，问题在于如何让 Vue.js 能够处理输入框中输入的信息，这里可以通过数据的绑定来解决。在传统的基于 DOM 的开发模型中，开发者需要选取 DOM 节点，获取节点中的数据，存放在逻辑层。当逻辑层的数据发生改变时，再手工选取 DOM 节点进行同步，而 Vue.js 这类的 MVVM 框架则能简化操作。

(2) 在 Vue.js 中，开发者可以通过“`v-model`”指令将数据绑定在 `Input` 元素和 Vue.js 的 `data` 属性中，一旦输入框中的数据发生了变化，同步更新属性 `data`。

输入框 HTML 部分，代码如下：

```
<input v-model="newTodo">
```

// 通过 `v-model` 指令绑定到 `data` 上

JavaScript 部分，代码如下：


```

var app = new Vue({                                     // 创建一个Vue实例
  data: {                                              // 实例的 data 属性管理数据信息
    newTodo: ''                                       // newTodo 数据默认为空字符串
  }
})

```

(3) 当用户输入完一条新备忘信息后，需要将该信息放入列表。这里设计通过单击回车键来进行存储过程，所以需要监听回车事件。被存储的数据结构除了备忘信息本身外，还需要为每条数据添加一个唯一 ID 标识和一个完成状态。HTML 部分，代码如下：

```
<input v-model="newTodo" @key.enter="addTodo"> // 添加了一个 key 的 enter 事件 addTodo
```

JavaScript 部分，代码如下：

```

var app = new Vue({
  data: {
    todos: [] // todos 对象存放备忘录信息
    newTodo: ''
  },
  methods: {
    addTodo: function() { // 添加新条目方法
      var value = this.newTodo && this.newTodo.trim() // 新条目（去除前后空格）
      if (!value) { return } // 如果新条目没内容，结束
      this.todos.push({ // 存放新条目到 todos 对象
        // id 增加 1，id 用来存 key
        id: this.todos.length ? this.todos[this.todos.length - 1].id++ : 1,
        title: value, // 存放新条目内容
        completed: false // 时候完成（默认没完成）
      })
      this.newTodo = '' // 将 data 里的新条目设空
    }
  }
})

```

(4) 在每条备忘录 HTML 结构中增加“删除”按钮，实现单条备忘的删除功能。HTML 部分，代码如下：

```

<ul> // 备忘录列表
  // v-for 为 Vue 指令，用来循环数组或者对象，key 存放 id，因为 Vue2 采用 Virtual Dom，
  // key 可以用来处理在重排时的列表项复用，推荐使用来提高性能
  <li v-for="todo in todos" :key="todo.id">
    <div>
      // checkbox 的 v-model 会默认用来绑定 true 或者 false 信息

```

```

        // label 用来展示条目的内容
        <input type="checkbox" v-model="todo.completed">
        <label>{{ todo.title }}</label>
        <button @click="removeTodo(todo)"></button>
    </div>
</li>
</ul>

```

JavaScript 部分，代码如下：

```

methods: {
  removeTodo: function() {
    this.todos.splice(this.todos.indexOf(todo), 1)
  }
}

```

// 增加一个 removeTodo 方法
// 删除 todos 下相应的条目

(5) 之前编写的 HTML 和 JavaScript 代码还处于分离状态，需要让实例选择一个挂载的节点去接管 HTML 上定义的指令和数据。HTML 部分，代码如下：

```

<section class="todoapp">...</section>

```

// 增加一个外层节点，绑定 Vue 实例

JavaScript 部分，代码如下：

```

app.$mount('.todoapp')

```

// 将实例挂载到 .todoapp 的 DOM 节点上

从这个例子中可以看出，Vue.js 的核心开发理念，在于让开发者尽量避免对于 DOM 的直接操作，而采用直接操作数据。至于数据的观察计算和绑定这些工作则由框架完成，而实际在 Vue.js 的核心代码中，对这些操作进行了大量的优化，在 Vue.js 2.0 中还引入了 Virtual DOM 来提高对 DOM 操作的性能，使其能够成为市面上性能最好的前端框架之一。

至此，一个简单的备忘录已然完成，样式部分可以通过引入标准的 TodoMVC 模板来实现。不过，一个完整的备忘录还需要具备能对每条备忘录进行编辑，过滤展示“完成”和“未完成”状态的条目等功能，具体的代码请参照本章源码。

9.3.2 管理应用的状态及实现组件间的通信

1. 简单状态管理

状态可以简单地理解为是 Vue.js 中的某些 Data 属性，这些数据是可以变化并且影响表现状态的。当组件数量增多时，整个应用的状态分散在各个组件或实例中，并且多个组件或实例可能存在状态共享，因此状态管理成了一个比较复杂的工作。

在 Vue.js 中，状态的管理有多种方式。在一个复杂度可控的环境下，可以通过 Vue.js 自身的

机制解决该问题。解决的原理是当访问数据对象时, 一个 **Vue** 实例只是简单地代理访问, 而不是由实例拥有这个数据对象, 这也意味着多个 **Vue** 实例可以引用共同的数据对象。

实现一个简单状态管理只需要将组件或实例中的公共部分存入一个单独的对象中, 各组件或实例在 **data** 属性中引用该对象即可, 实现代码如下:

```
var shareData = {}; // 共享数据
var vmA = new({
  data: shareData // 实例 vmA 中引用共享数据
})
var vmB = new({
  data: shareData // 实例 vmB 中引用共享数据
})
```

这样两个实例共享相同数据源, 当数据源发生改变时, 两个实例同时自动更新引用的视图。

上述只是一个最简单的实现, 复杂情况需要使用 **Store** 模式。一方面需要让组件或实例可以拥有和管理自己的私有状态, 另外一个重点在于, 当前模式对 **sharedState** 的修改并没有任何的记录 (属于隐式修改), 同时还允许在子组件中通过 **this.\$root.\$data** 访问或者修改数据源, 而一旦层级过深, 完全无法跟踪这种变化。现在, 通过 **Store** 模式解决该问题, 代码如下:

```
var store = { // 全局 store
  state: { count: 1 }, // 定义一个数据属性
  addCount (num) { // 定义一个方法改变数据
    console.log('增加 count 值')
    this.state.count = this.state.count + Number(num)
  }
}
var vmA = new Vue({
  data: {
    privateState: {}, // 实例 vmA 的私有属性
    sharedState: store.state // 实例 vmA 引用的共享属性
  }
})
var vmB = new Vue({
  data: {
    privateState: {},
    sharedState: store.state
  }
})
```

在 **Store** 模式下, 如果需要增加共享的状态 **count** 值, 需要调用 **addCount** 方法。当所有修改都

通过方法实现时，可以记录到哪种状态变化将会发生，如何被触发。当需要调试时，只需要在方法内进行处理，就可以方便地跟踪问题的上下文，实现了状态的统一管理。

简单状态管理可以处理大部分的实际需求，但是这种方式因为数据仍然可以被任意组件随意修改，更好的方法是组件通过分发事件通知 Store 进行改变，这也催生出了 Vuex 这类基于 Flux 思想的状态管理方式。

2. 组件间通信

组件是一种对可复用元素的封装，下面通过一个例子来学习 Vue.js 中组件间通信。HTML 部分，代码如下：

```
<div id="example">
  <my-component></my-component>           // 父组件调用子组件
</div>
```

JavaScript 部分，代码如下：

```
Vue.component('my-component', {           // 注册组件
  template: '<div>这是一个自定义组件</div>' // 组件的 template
})
new Vue({
  el: '#example'                         // 挂载实例在节点上
})
```

渲染结构部分，代码如下：

```
<div id="example">
  <div>这是一个自定义组件</div>
</div>
```

当应用被拆分成组件时，组件之间需要协同工作。在 Vue.js 中推荐开发者定义一个根组件，所有组件都可以视为父子组件关系中的一部分。共享状态是一种组件间通信的方法，除此之外，还提供了 Prop 和自定义事件处理父子组件间的通信。

Prop 是父组件向子组件传递的信息，默认为单向绑定，当父组件发生属性变化时，Prop 的数据会被修改，而子组件不能通过修改 Prop 来改变父组件的状态（在 Vue 1.0 中，可以通过 twoWay 关键字变成双向绑定，在 Vue 2.0 中被废除），父组件可以使用事件接口接收子组件的数据传递。

Prop 在子组件中的定义类似于 Data 属性。不仅需要在子组件中定义 Props 对象，同时还需要在父组件模板中加入对应的属性名引用，实例代码如下：

```
<my-component message="hello"></my-component> // 父组件模板修改
```



```
props: ["message"] // 子组件中添加一个 props 对象
```

当子组件向父组件传递信息时，注册一个 `methods` 方法，代码如下：

```
methods: {
  sendMsgToParent: function() { // 在 methods 中注册一个方法
    this.$emit('listenToChild', '子组件信息') // 通过 emit 传递信息到父组件
  }
}
```

再在子组件的模板中加入一个能够触发该方法的按钮，代码如下：

```
<button v-on:click="sendMsgToParent">传递信息给父组件</button>
```

父组件中添加响应方法监听自定义事件，并在 `methods` 中注册该响应事件，代码如下：

```
<p>{{ msg }}</p>
// 父组件通过 v-on 监听子组件的 emit 信息，并通过 showMsgFromChild 处理
<my-component v-on:listenToChild="showMsgFromChild"></my-component>
data: {
  msg: ''
}
methods: {
  showMsgFromChild: function(data) { // 方法接收到 emit 信息的 data
    this.msg = data; // 把信息赋值给父组件的 data，显示在模板上
  }
}
```

Vue.js 父子组件通信方式如图 9.7 所示。

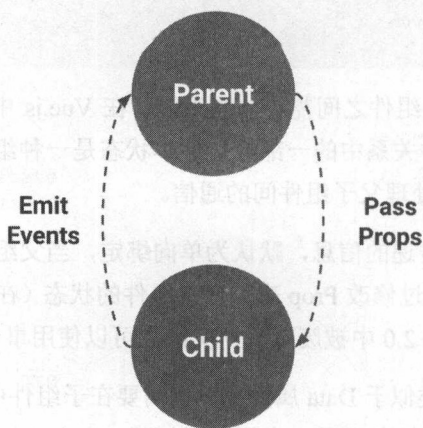


图 9.7 Vue.js 父子组件通信方式

Vue.js 2.0 对组件间的通信更为严格，一方面是 Prop 的严格单向绑定，另一方面废除了属性“\$dispatch”和“\$broadcast”这类允许组件自由交流的事件中心，这两方面都是为了防止开发者使用大量不透明的通信方式导致组件结构变得混乱而脆弱。非父子组件可以参考 Vue.js 官网使用自定义事件实现 Bus 模式，该模式类似于 Store 模式，通过给每个组件引入一个 Bus，在该对象上使用“\$on”和“\$emit”实现通信。对于简单的模型，上述的两种方式已经足够处理状态和通信，复杂的模型使用 Vuex 是一个更好的选择。

9.3.3 添加动画效果

Vue.js 不提倡直接操作 DOM，但实际开发中为了给用户带来更流程的使用体验，会在 DOM 变化时附带动画效果。Vue.js 提供了一些内置的方法来解决这类需求。

1. Transition 组件

Transition 组件是 Vue.js 提供的用于处理元素添加进入或者离开的过渡动画。将元素封装至 Transition 组件后，如果元素需要插入或删除，将依据以下顺序进行操作。

- 判断 Transition 组件上是否设置了动画，在相应的时机通过控制 CSS 类名实现动画效果；
- 判断 Transition 组件上是否设置了 JavaScript 钩子函数，在相应阶段调用钩子函数；
- 如果没有任何设置，在下一帧执行 DOM 操作。

Transition 组件可以为条件渲染、条件展示、动态组件和组件根节点添加动画效果，实例代码如下：

```
<transition name="fade">                                /* transition 组件用来包装元素并且命名 name */
  <p v-if="show">hello</p>                                /* 通过修改 show 的值控制元素的展示和消失 */
</transition>
.fade-enter-active, .fade-leave-active {                  /* 类名带上 name，控制进入的过渡动画 */
  transition: opacity .5s
}
.fade-enter, .fade-leave-to {                             /* 控制离开的过渡动画 */
  opacity: 0
}
```

从上述代码可以发现，在 Transition 组件上定义了一个 name 属性，该属性命名需要开发者在 CSS 代码的类名上添加前缀，Vue.js 会在实际渲染时修改成相应的类名。

2. CSS 类名

有 6 种 CSS 类名在进入或离开的过渡中切换，分别如下。

- **v-enter**: 进入过渡的开始状态。在元素插入前被添加，在元素插入后的一帧被移除。
- **v-enter-active**: 进入过渡的激活状态，在整个进入阶段都会生效。元素插入前被添加，在整个过渡或动画结束后被移除。
- **v-enter-to**: 进入过渡的结束状态，在元素插入的最后一帧被添加（同时 **v-enter** 被移除），当过渡或动画结束时被移除。
- **v-leave**: 离开过渡的开始状态，当一个离开过渡触发时立即被加入，在一帧后被移除。
- **v-leave-active**: 离开过渡的激活状态，在整个离开阶段都会生效，在过渡离开被触发时立刻被添加，当整个过渡或动画结束后被移除。
- **v-leave-to**: 离开过渡的结束状态，在离开过渡的最后一帧被触发（**v-leave** 被同时移除），在整个过渡或动画结束的时候被移除。

开发者可以通过这 6 类类名实现各个阶段的动画效果，在 CSS 文件中通过替换“v”为组件上的 name 完成对应关系。

3. JavaScript 钩子

在一些场景下 CSS 动画并不能满足需求，开发者需要额外使用 JavaScript。Vue.js 在 Transition 组件上提供了 8 个 JavaScript 钩子来调用 methods 中的函数，分别是 before-enter、enter、after-enter、enter-cancelled、before-leave、leave、after-leave、leave-cancelled，分别对应进入前、进入、进入后、进入被取消、离开前、离开、离开后、离开被取消的动作。这些动作有着一样的绑定方式，实例代码如下：

```
// HTML 代码，绑定了 before-enter 钩子到组件上
<transition v-on: before-enter="beforeEnter">...</transition>
// JavaScript 代码，定义 beforeEnter 的具体内容
methods: {
  beforeEnter: function(el) {
    ...
  },
  ...
}
```

这里需要注意的是，钩子 enter 和 leave 是一个异步耗时的动画过程，往往会持续一段时间，在参数中需要添加函数 done，并在函数中调用 done 来完成整个动画，否则会成为同步调用立即完成。实例代码如下：

```
enter: function (el, done) {
  // 添加第 2 个参数 done，手动完成整个动画
},
```

```
leave: function (el, done) {
  // 添加第 2 个参数 done, 手动完成整个动画
},
```

4. 多组件和列表过渡

当动画需要完成多个组件切换时, 可以使用动态组件, 实例代码如下:

```
// HTML 代码, 绑定 view 数据到 component 组件上实现动态切换组件
<component v-bind:is="view"></component>
// JavaScript 代码, 实现一个 view 的数据
data: {
  view: 'view-a'
},
// components 对应动态组件
components: {
  'v-a': { template: '<div>组件 A</div>' },
  'v-b': { template: '<div>组件 B</div>' }
}
```

使用动态组件可以跟普通元素一样, 通过 Transition 组件实现动画效果。另一种情况是列表过渡, 在 Vue.js 中一般使用“v-for”渲染一个列表, 这种情况下 Transition 无法做到同时渲染多个子元素, 因此提供了 Transition-Group 组件来解决这个问题。该组件会用一个真实的元素(默认为 Span 元素)进行渲染, 即用一个元素来为循环的节点提供一个外层元素来添加动画效果。在 Transition-Group 组件中也可以通过更改 Tag 属性为其他渲染元素, 同时需要给每个子节点提供一个唯一 Key 区分不同元素, 否则 Vue.js 为了效率只会提供相同标签的内部内容。Transition-Group 组件其他的使用方法同 Transition 组件一致。如果需要让移动更加平滑, 还可以使用 Transition-Group 的“v-move”特性, 在这里就不多赘述了。

9.4 打造单页应用 SPA

单页应用英文全称 Single Page Application, 简称 SPA, 近几年成为一个颇有热度的话题, 越来越多的网站使用这种模式进行网页开发, 事实上单页这个概念大概在 2003 年就已经被提出。单页应用的理念是所有需要的代码 (HTML、JavaScript、CSS) 在一次页面加载中获取, 或者部分资源在页面需要的时刻被动态加载。通过路径的 Hash 值或者 HTML 5 的 History API 提供的技术, 在单一页面内实现页面跳转切换, 但实际没有重新加载页面。

9.4.1 单页应用的优势是什么

单页应用主要依赖两种技术，一种是 JavaScript 的框架提供渲染和页面切换的能力，另一种是 AJAX 提供前端页面和服务器端数据交互的能力。AJAX 也可以被 WebSocket 等客户端与服务端通信技术替代，对常见的开发而言，AJAX 的使用更加普遍。

单页应用这个理念之所以如今越来越被开发者喜爱，主要的原因还是随着现代网站的开发复杂度和要求越来越高，开发者渐渐精细分工，出现了前端开发者和后端开发者角色。前端开发实现页面 UI、动画效果和部分页面逻辑，后端开发完成路由、页面数据绑定和另外一部分跟数据库交互的业务逻辑。精细化分工让前后端开发者能够集中精力完成自身专业部分，但同时也发现分工协作极大地增加了联调开发和沟通成本，随着网站迭代的频率越来越快，这种模式的开发成本也越来越难以接受。

单页应用模式意味着从渲染、交互逻辑到路由交由前端实现，后端仅需要提供数据接口，前端与后端的交互只停留在数据层，后端的开发完全不依赖于前端，前端也可以通过接口数据的模拟脱离后端实现整个开发流程。

单页应用的优势还体现在目前移动应用流行的大环境下，后端仅需要维护一套接口即可完成多端适配。在用户体验上，没有页面重载意味着用户在路由变化时只需要调用小部分接口获取数据，响应速度更快并且对于服务端的压力更小。

单页应用也有一定的缺点，一个最难以忽视的问题在于 SEO（搜索引擎优化）。因为所有的数据均由 AJAX 从后端接口获取，当搜索引擎爬虫抓取页面时并不执行 JavaScript，往往获得的是一个空的 HTML 的页面。对于这个问题，虽然有一些针对单页应用的 SEO 手段，但是开发者更应该使用单页应用实现不太需要考虑 SEO 优化的页面。

另外一个缺点是当开发大型单页应用时，最终的 JavaScript 文件体积会变大，并且数据获取发生在页面加载完成之后，期间产生一段白屏时间，这对用户的体验并不友好。针对这个问题，开发者需要对于单页内部的路由进行拆包，实现按需加载，当进入特定路由才加载该路由的 JavaScript 业务文件，减少首屏的加载时间。同时，服务器端还可以通过渲染一部分首屏页面数据，避免造成首页加载产生过多请求。

本节主要介绍了为什么需要单页应用以及单页应用的优缺点及应对措施，后面的章节开始具体讲述如何开发单页应用。

9.4.2 实战演练：实现一个单页路由

路由是一个单页应用的核心，大部分前端框架都实现了一个复杂的路由库，包括动态路由、

路由钩子、组件生命周期甚至服务器端渲染等复杂的功能。但是对于前端开发者而言，路由组件的核心是 URL 路径到函数的映射，了解了这个概念，便可以亲自实现一个简单的路由功能。

1. 路由的原理

路由 (Route) 在前端可以理解为 URL 路径到函数的映射。当访问到一个特定的路径时执行特定的函数。另一个概念 Router，用于管理各种 Route，也就是匹配路径到函数的过程。

2. 实现路由

Web 端实现路由有如下两种技术模式。

- 基于 Hash
- 基于 History API

Hash 路由的路径中会有一个“#”标志，即常说的锚点，前端向后端服务器发送请求时并不会解析 Hash 部分。路由实现通过监听页面 window 对象的 hashChange 事情，调用对应的函数，优点是兼容性好且完全脱离后端，缺点是因为带了 Hash 标志导致路由不直观。

History API 通过监听 HTML 5 添加的 popstate 事件实现路由，URL 格式跟传统的后端路由一致，这也是这种模式最大的优点。缺点是只有新型浏览器支持该特性，且需要后端路由配合，因为当用户访问一个 History 路由实现的路径时，页面仍会向后端请求，如果后端没有设置相应的路由实现逻辑，将返回 404 错误。

根据前端路由的概念，实现一个路由需要三个部分：存储路径和对应的回调方法、监听浏览器的相关事件，根据监听结果执行路径对应回调方法。根据这些要求，在开发的路由模块中，设计一个对象通过“Key-Value”模式存放路径和对应方法，再通过 window 对象监听 popstate 事件，根据当前的路径从路由对象中选择执行对应的方法，实现的代码如下：

```
function Router() {
  this.routes = {}; // 存放路径和对应方法
  this.route = function(path, callback) { // 实例化后通过调用来增加新的路由
    this.routes[path] = callback; // 通过 key-value 存放 callback
  }
  this.refresh = function() { // 通过一个函数调用最终的 callback
    var curUrl = location.hash.slice(1) || '/'; // 在 Hash 模式下获取路径
    // var curUrl = location.pathname; // 在 History API 模式下获取路径
    this.routes[curUrl](); // 调用最终的 callback
  }
  this.init = function() { // 初始化方法
    // 监听 load 事件对应第一次页面加载
```

```

window.addEventListener('load', this.refresh.bind(this), false);
// Hash 模式下监听 hashchange 事件
window.addEventListener('hashchange', this.refresh.bind(this), false);
// History API 模式下监听 popstate 事件
// window.addEventListener('popstate', this.refresh.bind(this), false);
}
}

```

在实际开发过程中, 需要调用 `route` 方法添加路由和对应方法, 代码如下:

```

var router = new Router(); // 实例化 Router 方法
router.init(); // init 来监听对应的全局事件
router.route('/', function() { ... }); // 用过 route 方法添加新的路由和对应方法
router.route('test', function() { ... });

```

这样就实现了一个功能简单的单页路由, 对于使用了 `React` 或者 `Vue.js` 这样的复杂单页应用, 路由组件还实现了一系列复杂的功能。本节只实现了一个简单的路由模块, 其他复杂的功能可以参考 `React Router` 或者 `axios` (`Vue 2.0` 推荐路由) 的源码。

9.4.3 实战演练: 使用 React 开发一个简单的单页应用

`React` 是一个专注 `View` 层的前端框架, 使用 `React` 开发一个单页应用推荐采用官方维护的 `React Router`, 地址为 <https://github.com/ReactTraining/react-router>。

在使用前, 先安装 `React Router`, 命令如下:

```
npm install react-router -save
```

`React Router` 能够很好匹配 `React` 的最大原因是继承了组件化思维。在 `React` 中组件即方法, `Props` 即参数。在 `React Router` 中, `Router` 可以看成是一个 `React` 组件, 路径作为参数, 返回相应的 `View`。

`React Router` 的使用方式和 `React` 一样, 也是一致的声明式渲染模式, 开发者不用关心内部细节, 通过 `JSX` 语法声明式的调用 `Router` 组件。下面是一个最基本的例子, 代码如下:

```

import { render } from 'react-dom';
import { Router, Route } from 'react-router';
import App from '<project-path>/containers/App';
const routes = {
  <Router> // Router 组件管理路由
    <Route path = "/" component = { App } /> // 通过 props 定义 path, 匹配时调用组件
  </Router>
}

```

```

}
render(routes, document.body);

```

Router 组件作为最外层组件，根据路径判断渲染对应 Route 节点。如上实例，当路径匹配到“/”时，调用 App 组件进行渲染。Router 允许添加多个 Route，依据不同的 Path 去展示不同的组件。

• 嵌套路由

嵌套路由是某个路径的子路径，嵌套路由跟平级路由的差别不仅仅只在路径的层级，也在于组件的复用。通常路由的嵌套意味着页面上存在公共部分，嵌套路由可以根据路由的层级拼装多个组件在页面上展示，下面是一个嵌套路由的例子，代码如下：

```

<Router history={ browserHistory }>                                // 同步浏览器路由
  <Route path = "/" component = { App } />
    <IndexRoute component = { Dashboard } />                        // 子路由，路径一致
    <Route path = "inbox" component = { Inbox } >                  // 子路由，路径为下一级
      <Route path = "/messages/:id" component = { Message } />
    </Route>
</Router>

```

在这个例子中 App 组件通常不能完整渲染整块页面，需要借助 Props 中的 Children 属性，下面是一个 App 组件的实例，代码如下：

```

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>App</h1>
        { this.props.children }    // 用来渲染子组件的内容
      </div>
    )
  }
}

```

单独的 App 组件无法完成整个页面的渲染，因为其中调用了子组件的内容。这里可以使用“或”语句处理只在根路径的渲染，也可以使用 React Router 提供的 IndexRoute 方法，当路由进入父组件的路径时，调用 IndexRoute 中绑定的组件。

React Router 还支持路由中附带参数，比如，符合路由规则的“/messages/:id”，具体的“id”值在组件中可以通过 props.params.id 来获得。在上面的实例中，实际渲染组件见表 9.2。

表 9.2 实例中渲染组件对应列表

URL	Components
/	App -> Dashboard
/inbox	App -> Inbox
/inbox/messages/:id	App -> Inbox -> Message

- 多种路由模式

React Router 支持 Hash 路由和 HTML 5 的 History API。HTML 5 的 History API 需要引入 browserHistory，而使用 Hash 路由时需引入 hashHistory，代码如下：

```
import { Router, Route, browserHistory } from 'react-router';
...
<Router history = { browserHistory } />
...
</Router>
```

- 路由的组件生命周期

React Router 实现的一个单页应用，由于每一个路由都可能存在父子组件，因此会存在组件生命周期变化的过程。当用户打开根节点路由时，根节点和 IndexRoute 对应的组件完成了 componentDidMount 过程。当根节点跳转到具体路由时，IndexRoute 组件因为不再被渲染，会调用 componentWillUnmount 上的方法，根节点组件则因为更新了子组件从而触发 componentWillReceiveProps 和 componentDidUpdate 两个生命周期方法，而具体路由上的组件则调用 componentDidMount 方法。

当同一级路由下参数变化时，会调用父子组件的 componentWillReceiveProps 和 componentDidUpdate 生命周期方法，因为所有组件都已经被挂载，同一级路由切换时生命周期的变化情况，与从根节点切换到具体路由中 IndexRoute 或新路由的情况一致，此时，旧路由上的组件都会调用 componentWillUnmount 的方法。

了解路由变化的组件生命周期，开发者可以更好地决定何时在父子组件生命周期中处理对应逻辑，通常来说，需要关注的事件有 componentDidMount、componentDidUpdate、componentWillUnmount 和父组件中的 componentWillReceiveProps。

9.4.4 单页应用的状态管理

本节将继续探究使用 React Router 的单页应用维护整个单页应用的状态。Redux 已经成为维护一个大型 React 应用必备的状态管理工具，开发者自然会想到使用 Redux 和 React Router 来完成整

个单页应用的状态管理。的确在 React 中，单页应用的本质是完成组件间切换，但 URL 也是状态管理中的一部分，而 Redux 并不能收集到这一信息。

在使用 Redux 构建应用时，开发者需要确保应用状态的单一原则，即统一的 Store 管理，因此路由状态也需要被包含在内，所以最终 React 提供了“react-router-redux”库来实现这一功能，GitHub 地址为 <https://github.com/reactjs/react-router-redux>。

注意：并不是所有的情况开发者都需要“react-router-redux”，只有在程序使用 Redux 进行管理状态，并且需要业务逻辑与路由状态有较强的关系。例如，在路径中的分页和排序参数或者是需要记录、持久化、重放用户的行为、行时间旅行时，才需要进行使用。

1. 绑定 React Router 和 Redux

首先通过 NPM 安装“react-router-redux”，命令如下：

```
npm install --save react-router-redux
```

库本身提供了一个 syncHistoryWithStore API 完成和 Redux 的绑定，需要的参数是 React Router 的 History 和 Redux 的 Store。执行 syncHistoryWithStore 返回一个新的 History 对象，将这个对象作为 Props 传递给 Router 组件，再通过 Redux 提供的 Provider 组件在最外层接管整个 Store，完成两者的绑定，代码实例如下：

```
import { browserHistory } from 'react-router';
import { syncHistoryWithStore } from 'react-router-redux';
import reducers from '<project-path>/reducers';
const store = createStore(reducers);
// 通过 syncHistoryWithStore 方法将生成的 store 同 browserHistory 绑定
const history = syncHistoryWithStore(browserHistory, store);
```

2. 访问路由状态

React Router 通过 Route 组件的 Props 提供路由信息，开发者可以方便地在容器型组件中访问这些数据。当使用 React Redux 的 connect 方法连接组件到全局状态时，可以通过 mapStateToProps 函数的第二个参数访问路由信息，实例代码如下：

```
function mapStateToProps(state, ownProps) { // mapStateToProps 将作为参数被 connect 使用
  return {
    id: OwnProps.parms.id,           // 因为绑定能从第二参数上获得路由信息
    filter: OwnProps.location.query.filter
  };
}
```

因为 React Router 采用异步运行（处理例如动态加载组件），组件树可能不与 Redux 的状态同步更新，所以在 Redux Store 中直接读取 location 状态是不适用的。而通过 React Router 传递 Props 中获取的 location 状态是处理了所有异步代码更新后的结果。

3. 通过 Redux 改变路由

在 Redux 中想要改变状态，必须要分发一个 Action，所以通过 Redux 来改变路由也需要做同样处理。在分发之前开发者需要在 Redux 的 Store 中绑定 History 对象，“react-router-redux”提供了 routerMiddleware 这样一个 middleware 工厂中间件，通过传入 History 对象，生成一个真正的 Redux 中间件。通过使用 routerMiddleware 中间件，当开发者通过 store.dispatch 执行路由跳转时，中间件会调用 History 对象实现路由变动。下面是一个实现的实例，代码如下：

```
import { createStore, combineReducers, applyMiddleware } from 'redux';
import { routerMiddleware, push } from 'react-router-redux';
const middleware = routerMiddleware(browserHistory); // 获得 history 参数生成中间件
const store = createStore(
  reducers,
  applyMiddleware(middleware) // 绑定中间件到 dispatch 过程中
);
// 具体组件通过 Redux 来 dispatch 路由
import { push } from 'react-router-redux';
store.dispatch(push('/foo'));
```

9.5 本章小结

随着整个前端生态圈的繁荣发展，更多先进的设计理念被融入前端框架中。作为其中佼佼者的 React 和 Vue.js 已被更多的开发者所熟知，而传统的基于 DOM 操作的编程方式，在处理复杂的前端应用上则略显单薄。本章首先从基于 DOM 编程方式在移动端的应用着手，逐步介绍 React 及 Vue.js 的基本编程思想以及实战应用，希望对读者有所启发。当然，任何优秀的框架都为需求服务，都有其更为适用的业务场景，在处理实际问题时，也应从实际情况出发，选择更为合理的编程方式。

10

第 10 章

混合式开发实战

混合式开发的理念综合了原生和 Web 开发的特性,通常定义为开发一个在原生容器使用 Web 技术的混合式应用。原先的混合式开发,在理念上保持着使用 HTML、CSS、JavaScript 的体系结构在 WebView 中执行的概念,但在现阶段,Web 前端技术开始更深入地融入原生应用的开发体系。所以本章使用了混合式开发作为标题,一切使用 Web 技术来开发原生移动应用的场景都可以视为混合式开发。

10.1 为什么需要混合式开发

本节先来介绍混合式开发有哪些,它们有什么优势,我们该如何选择。

10.1.1 混合式开发种类

1. WebView 模式

WebView 模式的代表是 PhoneGap 和 Cordova,APP 的原生部分仅仅作作为一个容器,主要的

业务代码都通过 HTML、CSS、JavaScript 放置在 WebView 中执行。

2. JavaScriptCore 模式

除了传统的基于 WebView 的混合应用, JavaScriptCore 带来了一种全新的开发模式, 即通过 JavaScript 调用原生代码渲染原生控件的混合式开发。

JavaScriptCore 框架原来只提供在 WebView 的 WebKit 内核中, 在 iOS 7 中开放了这一能力, 之后在 Android 中也提供了类似的功能, 从而催生出了 React Native 这样使用 JavaScript 作为 Bridge 操作原生代码来构建应用的方案, 构建出的整个应用可以被理解为原生应用。

3. 微信小程序

微信小程序本质仍使用了 WebView 方案, 但独立设计了一套语法对应传统的 HTML、CSS 和 JavaScript, 限制了一些可能导致问题的语法, 在部分组件上也学习了 React Native 类框架直接渲染原生组件提升性能, 再利用离线缓存获得流畅的体验。这种方案依赖应用的内部设计, 并非一个通用的解决方案。

4. Flutter

除了以上几种方案, 谷歌还推出了 Flutter 混合式跨平台方案, Flutter 更激进地实现了整个 UI 层, 可以通过 Dart 语言直接控制完成。这里提到 Flutter 是因为 Flutter 整体的 UI 界面编写类似于一种声明式的 HTML 和 CSS 混编语法, 并且谷歌曾希望 Dart 能够替代 JavaScript 在浏览器中的地位, 因为这种方案还处于预览阶段且与 Web 前端开发差异较大, 这里不多赘述。

10.1.2 混合式开发的优势

混合式开发虽然让 Web 前端工程师能够融入原生应用的开发流程中, 但初衷并非仅仅如此, 混合式开发存在着几个显著的优势让其能够被各大厂商迅速接受, 主要的优点有以下三个。

1. 跨平台

在不同的移动平台一般使用不同的编程语言, 提供不同的 API 和不同的 UI 编写方法。大多数应用都需要开发多套不同的代码来应对, 而混合式开发的多数业务逻辑都使用了各平台支持的通用语言进行开发, 且 HTML、CSS 和 JavaScript 相对于别的混合式开发语言入门门槛更低, 能有效地减少开发人力成本。

2. 快速发布

传统 APP 的分发依赖于应用市场, 而大多数应用市场存在着审核机制, 一个 APP 的更新迭代或者 Bug 修复从开发到触达用户会长达一周甚至数周。而原生应用需要编译过程, 在 Android 端

虽然有一定的动态更新能力，但是往往有诸多限制。

Web 代码无须编译，目前也是各大主流平台允许的热更新方案，所以对于应用来说，一些活动或者经常发生功能或 UI 变动的页面是采用混合式开发的合理场景。

3. 功能提升

如果说只是需要前两个优点，WebView 加载页面就能够很好的处理，然而混合式开发还需要原生端的支持。一方面 Web 端在 API 能力上有限制，无法充分利用原生端的资源，而混合式开发可以将这些能力暴露给 Web 端调用。混合式开发的意义同样也在于获得原生端的优势，让用户能够拥有更好的体验。

10.1.3 选择合适的混合式开发方案

对于开发者，现有的混合式开发方案多种多样，选择一个适合的方案需要综合三点：开发效率、用户体验和项目复杂度。

如果开发者已经有了现成的应用并且有一定的原生代码开发能力，只是需要让 Web 页面嵌入 APP 中实现一些动态展示，可以选择自行实现一套 WebView 扩展方案，将原生能力暴露到 WebView 中，这种方案轻量且对现有应用影响很小。

如果内嵌的 H5 页面比较复杂或者是需要构建一个完整的 APP，且开发者更专长于 Web 技术，适合 PhoneGap 或 Cordova 这类的方案。其中，Cordova 提供了强大的插件系统，开发者可以在不熟悉原生代码的前提下快速获得原生应用的能力。需要注意的是，在某些场景下（如动画），WebView 的表现并不流畅，通过这种方案构建的完整应用体验较差。

另一方面，开发者需要注意到 WebView 在 Android 系统中存在碎片化，即会出现机型表现差异性，可以引入 Crosswalk 插件替代原有系统的 WebView 来保证功能或样式的一致性。在 iOS 平台中也可以引入 WKWebView，提升整体的性能、稳定性和丰富功能。不过，引入一个新的 WebView 会增大自身 APP 的体积，需要开发者权衡利弊。

对于 React Native 或 Weex 这类技术，既可以实现完整的应用也可以嵌入 APP 作为其中的一部分。这类技术能够提供接近原生应用的体验，但是也存在一些问题，主要体现在：一方面会存在一些平台兼容问题，另一方面 JavaScript 和原生代码涉及到多进程的异步通信模型，造成部分效率损耗，在复杂场景下存在一定的性能问题。除此之外，因为整个运行过程对原生代码依赖较重，出现问题较难排查，需要开发者掌握一定原生开发和排查问题能力。此类技术是目前混合式开发的主流前进方向，开发者应当在保持一定技术敏感的前提下，学习并视情况使用。

10.2 Cordova 开发入门

Cordova 是一个基于 WebView 下的 Web 和 Native 代码交互混合式开发模式。本节会先介绍 JavaScript 和 Native 代码互相调用的方式，在相互调用的模式之外，Cordova 的开发和纯粹的 Web 开发基本类似，并没有太高的学习成本。学习 Cordova 可以快速入门混合式开发，而在深入学习之后，开发者也可以通过 Cordova 插件来实现在 Native 端的无限可能。

10.2.1 JavaScript 和 Native 互相调用

混合式开发的本质是 JavaScript 代码和 Native 代码的互相调用，在这点上每个系统都有自己不同的使用方式。学习相互调用的方式，可以让开发者更好地了解调用原理，也能根据自身的需求扩展暴露的方法。

本节将简单介绍 JavaScript 和 Native 互相调用过程，因为这部分涉及原生知识，如需完整了解整个过程，可以参考 Android 和 iOS 相关开发文档。

1. Native 调用 Web

在 Android 和 iOS 中都提供了原生 JavaScript 解释器来执行 JavaScript 代码，使用也并不复杂，两个平台调用实例如下。

Android 端中 Java 实现，代码如下：

```
webView.loadUrl("javascript:(function(){alert('调用来自 Native');})();")
```

iOS 端中 Swift 实现，代码如下：

```
webView.stringByEvaluatingJavaScriptFromString(from: "alert('调用来自 Native')")
```

这是最普遍的实现方式，iOS 目前的新版本中提供了 WKWebView，调用方式与 UIWebView 接近，代码如下：

```
webView.evaluateJavaScript("alert('调用来自 Native')") {(item, error) in
    // 闭包中处理执行成功或错误的回调
}
```

注意：在 WKWebView 中，前端 window 对象里的 alert、confirm 和 prompt 方法在调用时不会直接显示，需要在 WKUIDelegate 中进行重写。

最后是 JavaScriptCore 的使用，JavaScriptCore 支持配合 UIWebView 使用，也可以在没有 UIWebView 的情况下使用。JSContext 是 JavaScriptCore 中的 JavaScript 执行环境，使用 Swift 代码

实现如下:

```
let context: JSContext = JSContext()
context.evaluateScript("1+1") // 输出 2
```

2. Web 调用 Native

Native 调用 Web 是 JavaScript 脚本的动态执行, 而 Web 调用 Native 则是原生获得 JavaScript 端数据, 然后执行或映射到原生代码的过程。所以相对 Native 调用 Web 复杂一些, 且需要 Web 端和 Native 端进行约定, 是否允许 Web 端调用 Native 端的任意代码。

Android 端存在三种方法:

- 第一种是重写 `WebViewClient.shouldOverrideUrlLoading`, 在页面中的 URL 变化时, 该函数会被触发, 通过将参数封装在 URL 内可以让 Native 响应解析然后执行相应的 Native 方法。
- 第二种是重写 `WebChromeClient.onJsPrompt`、`onJsConfirm`、`onJsAlert`, 但 Web 执行相应的 `window` 方法时会被触发。
- 最后一种是目前最主流的方法, `WebView.addJavascriptInterface`, 这种方法能将 Java 中的对象映射到 JavaScript 中, 调用 JavaScript 对象下的函数时, 会触发原生对象下的函数, 代码如下:

```
webView.addJavascriptInterface(new Object() {
    @JavascriptInterface
    public void funcTest() {
        // 需要执行的内容
    }
}, "androidObject");
```

这样在 JavaScript 中使用 `androidObject.funcTest` 方法就能执行原生端的方法。

注意: 从 Android 4.2 开始, 使用 “@JavascriptInterface” 注解的公共方法才会暴露给 JavaScript, 防止通过反射手段在 Web 调用一些不慎暴露出来的方法。

iOS 端在 `UIWebView` 中通过 `shouldStartLoadWithRequest` 实现类似 Android 中 `shouldOverrideUrlLoading` 的方式。

JavaScriptCore 提供原生代码被 JavaScript 调用需要先定义遵守 `JSExport` 协议的 `JavaScriptDelegate` 协议, 代码如下:

```
@objc protocol SwiftJavaScriptDelegate: JSExport {
```



```
...    // 具体方法
}
```

然后通过一个模型实现 `SwiftJavaScriptDelegate` 协议，代码如下：

```
@objc class SwiftJavaScriptModel: NSObject, SwiftJavaScriptDelegate {
    weak var controller: UIViewController ?
    weak var jsContext: JSContext?
    ...    // 具体方法
}
```

最后通过 `WebView` 加载页面，并在 `webViewDidFinishLoad` 代理中将定义的模型注入到页面中。

说明：`WKWebView` 中的调用方式和 `UIWebView` 基本一致，但是单独提供了方法“`delegate: WKScriptMessageHandler`”进行通信约定，因为 `WKWebView` 的应用场景还比较少，所以这里不做展开，有需要的读者可以查找官方文档。

10.2.2 Cordova 介绍和安装

1. Cordova 介绍

Cordova 是一个基于 `WebView` 的跨平台解决方案，其核心组成部分是插件，插件提供了 Cordova 和原生组件相互通信的接口并绑定到了标准的设备 API 上，使开发者可以通过 JavaScript 调用原生代码。

Cordova 的默认项目不存在任何插件，开发者可以根据自己的需求定制化地添加插件，插件的模式保证了项目不会引入不需要的功能，减少最终应用的包体积。插件同时也支持第三方开发来丰富 Cordova 的整个生态，需要注意的是，开发插件需要熟悉相关平台的原生代码。

基于 `WebView` 的解决方案虽然在性能和兼容性上存在一些问题，但是表现出了最强大的跨平台能力，Cordova 可以在一套 Web 代码的基础上迅速完成多平台的应用开发，如图 10.1 所示。

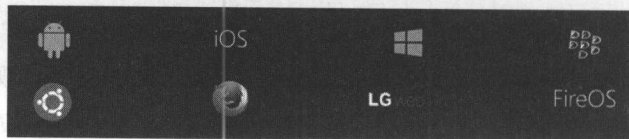


图 10.1 Cordova 支持的系统

2. Cordova 安装

Cordova 依赖于 Node.js，读者可以参考“第 2 章 移动 Web 开发环境搭建”的内容，通过 NPM

安装 Cordova，命令如下：

```
sudo npm install -g cordova          // OS X and Linux
npm install -g cordova              // Windows
```

创建项目，命令如下：

```
cordova create hello com.example.hello HelloWorld
```

添加平台，命令如下：

```
cordova platform add ios
cordova platform add android
```

构建 Cordova 应用还需要相关环境支持，Android 平台推荐安装 Android Studio，并注意勾选安装 Android SDK 和 Android Device Emulator。iOS 平台目前只支持在 Mac OS 系统下开发，需要安装 Xcode 和其命令行工具。

Cordova 检查前置依赖，命令如下：

```
cordova requirements
```

执行后，能在控制台看到如下说明，如图 10.2 所示。

```
cordova requirements

Requirements check results for android:
Java JDK: installed 1.8.0
Android SDK: installed true
Android target: not installed
android: Command failed with exit code 2
Gradle: installed /Applications/Android Studio.app/Contents/gradle/gradle-3.2/bin/gradle

Requirements check results for ios:
Apple OS X: installed darwin
Xcode: installed 8.3.2
ios-deploy: installed 1.9.1
CocoaPods: installed
Error: Some of requirements check failed
```

图 10.2 Cordova 检查前置依赖项

根据说明完成需要的依赖安装，然后构建 APP，命令如下：

```
cordova build          // 为所有添加的平台构建
cordova build ios      // 为特定平台构建
```

测试 APP，命令如下：

```
cordova emulate android  // 在模拟器中测试
cordova run android      // 在默认设备中测试（在有真机设备插入电脑后会启用真机设备）
```

可能出现的问题如下。

(1) 在项目构建中提示 “Error: Could not find gradle wrapper within Android SDK”。

Android studio 安装的当前版本 Android SDK Tools (Version: 23.0.2) 和 Cordova 的默认 Android 模块不兼容, 需要升级 Cordova 的 Android 模块版本。升级命令如下:

```
cordova platform update android@6.2.2
```

(2) 在启动 Emulate 提示 “Error: android: Command failed with exit code 2”。

因为 Cordova 的构建脚本与当前 Android Studio 默认模拟器不兼容, 需要更改项目的 “platforms/android/cordova/lib/emulator.js” 文件。

第一处修改代码如下:

```
// 更改前
return superspawn.spawn('android', ['list', 'avds'])
// 更改后
return superspawn.spawn('android', ['list', 'avd'])
```

第二处修改代码如下:

```
// 更改前
var command = 'adb -s ' + target + ' install -r "' + apk + '"';
// 更改后
var command = 'adb uninstall "' + pkgName + '" ; adb -s ' + target + ' install -r "' + apk + '"';
```

3. Cordova 开发环境

Cordova 应用的主体是 HTML、CSS 和 JavaScript, 对比原生代码, Web 代码有着无须编译、所见即所得的优点。在 Cordova 中开发者也可以利用这一优点, 在开发过程中进行热重载。安装插件 BrowserSync, 命令如下:

```
cordova plugin add cordova-plugin-browsersync // 安装插件
```

插件集成了 BrowserSync 在 Cordova 的工作流中, 观察 www 目录并修改 index.html 的部分 Meta 标签属性, 代码如下:

```
// 修改<meta http-equiv="Content-Security-Policy" /> 中 content 属性的如下部分
// 更改前
default-src 'self' data: gap: https://ssl.gstatic.com
// 更改后
default-src 'self' data: gap: ws: 'unsafe-inline' https://ssl.gstatic.com
```

然后执行, 命令如下:

```
cordova run android -- --live-reload
```

在编辑器中修改 `www` 目录下的文件，保存之后模拟器会自动刷新页面，即可以实时看到修改的内容。

10.2.3 Cordova 开发使用

Cordova 的开发使用跟一般的 Web 差异很小，只通过插件对 `WebView` 进行能力扩展，架构如图 10.3 所示。

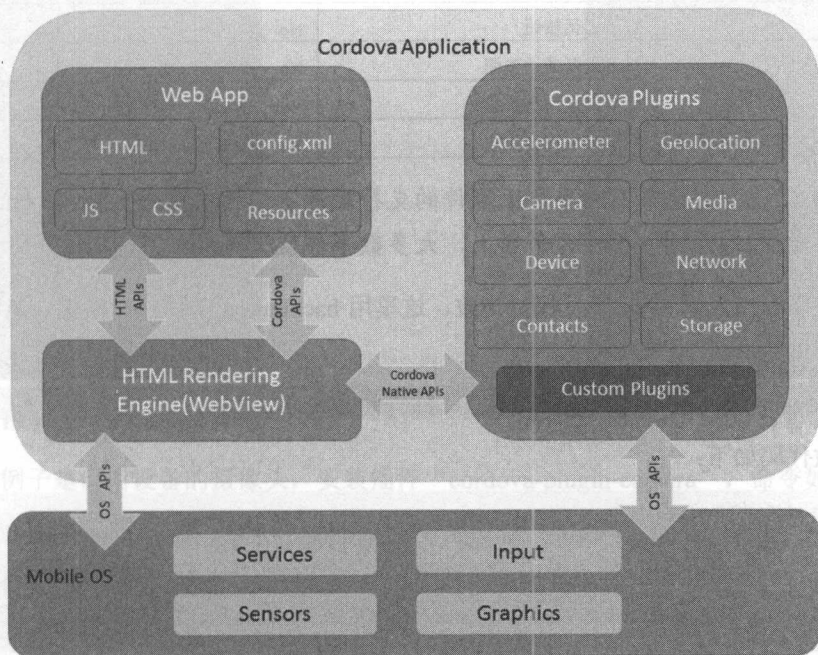


图 10.3 Cordova 架构图

Cordova 另外拓展了 `document` 对象上的一些监听事件，让开发者可以观察到设备的变化并进行响应，观察 Cordova 默认生成的 `index.js` 可以发现，几乎所有代码都需要在监听到 `deviceready` 事件后执行，即表示 Cordova 能力的注入是在 `WebView` 创建后。

对于 Web 前端开发人员，Cordova 开发主要需要关注插件 API 的使用和相关事件监听。

1. 事件

Cordova 主要支持的事件见表 10.1 所示。

表 10.1 Cordova主要事件

事 件	功 能	Android支持	iOS支持
deviceready	Cordova设置API准备好	Yes	Yes
pause	程序进入后台	Yes	Yes
resume	程序从后台回前台	Yes	Yes
backbutton	按下返回按钮	Yes	No
menubutton	按下菜单按钮	Yes	No
searchbutton	按下搜索按钮	Yes	No
startcallbutton	按下通话按钮	No	No
endcallbutton	按下挂断通话按钮	No	No
volumedownbutton	按下降低声音按钮	Yes	No
volumeupbutton	按下增加声音按钮	Yes	No

提示：需要注意的是在不同平台上事件的支持度是不同的，这里列出了 Android 和 iOS 平台的支持度。在 iOS 平台上，大多数事件监听不能使用。

事件的监听和回调与 Web 开发保持一致，这里用 backbutton 事件作为实例。监听 backbutton 事件代码如下：

```
document.addEventListener('backbutton', onBackKeyDown, false);
```

响应函数代码如下：

```
function onBackKeyDown(e) {
    e.preventDefault();           // 阻止默认事件发生
    alert('"后退按钮"被单击啦!');
}
```

使用者单击系统回退按钮时，不进行回退，并跳出弹窗，效果如图 10.4 所示。

2. 插件 API

在 Cordova 中开发者可以获取当前设备的信息，进行一些平台差异化的开发，首先需要安装“cordova-plugin-device”插件，命令如下：

```
cordova plugin add cordova-plugin-device
```

Cordova 的插件也需要通过监听 deviceready 事件确保被注入，代码如下：

```
document.addEventListener("deviceready", onDeviceReady, false);
```

在 onDeviceReady 函数中显示设备信息，代码如下：

```
function onDeviceReady() {
  alert(`platform: ${device.platform}; version: ${device.version}`);
};
```

设备上显示如图 10.5 所示。

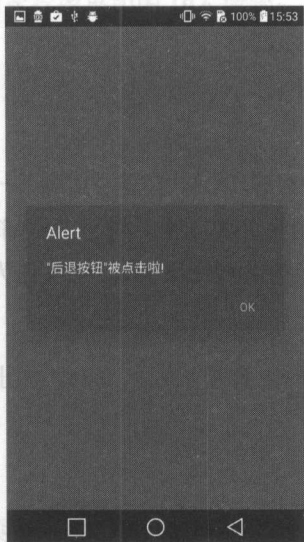


图 10.4 监听回退按钮实例

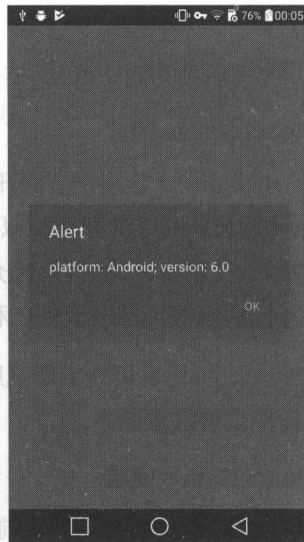


图 10.5 设备信息显示

另一个例子是调用设备的摄像头，安装插件“cordova-plugin-camera”，命令如下：

```
cordova plugin add cordova-plugin-camera
```

调用摄像头，代码如下：

```
navigator.camera.getPicture(cameraSuccess, cameraError, cameraOptions);
```

cameraOptions 是对摄像头调用的参数设置。一个比较通用的设置参数如下：

```
var cameraOptions = {
  quality: 50, // 设置质量，一般为 20、50 或 100
  destinationType: Camera.DestinationType.FILE_URI // 拿到的图片地址类型
};
```

一旦成功拍摄，就可以在 cameraSuccess 中拿到图片的存储地址，可以通过绑定到页面上的某个 IMG 标签的 src 属性来显示，代码如下：

```
function cameraSuccess(imgUri) {
  var image = document.getElementById('imgFile');
```

```
image.src = imgUrl;  
}
```

10.3 React Native 实战

10.3.1 React Native 简介

1. React Native 简介

React Native 是一套跨平台的开发解决方案, 基于 React 的虚拟 DOM 模型, 实现了一套代码多端运行。React Native 的开发类似于 React, 支持组件生命周期和 JSX 语法, 对于 Web 前端开发人员十分友好。在本章开始时, 已经介绍过 React Native 最大的优势是其使用 JavaScript 作为 Bridge 调用原生方法和组件, 兼顾了性能和开发效率。

目前 React Native 仍处于测试版本, API 并不稳定, 本书的内容基于版本 0.44, 如出现 API 不兼容, 需要读者注意版本。

2. React Native 安装

React Native 依赖 Python2(目前不支持 Python3 版本), 在 Mac OS 和 Linux 环境下无须安装, 在 Windows 下需要下载安装包。

(1) React Native 目前使用 Yarn 替代 NPM, 使用 NPM 在全局安装 Yarn, 命令如下:

```
npm install -g yarn
```

(2) 安装 React Native 命令行工具用于执行创建、初始化、更新项目、运行打包服务等任务, 命令如下:

```
npm install -g react-native-cli
```

(3) 在 Mac OS 环境安装 Xcode 及其命令行工具支持 iOS 开发, 其他系统暂不支持 iOS 开发。安装 Android Studio (包含 SDK 和模拟器) 提供 Android 开发环境, 推荐安装 Watchman 来观察文件改变进行刷新。

提示: Watchman 用于监听文件和记录文件的改动变化, 并在相关变化中触发命令动作, GitHub 地址为 <http://facebook.github.io/watchman/>。

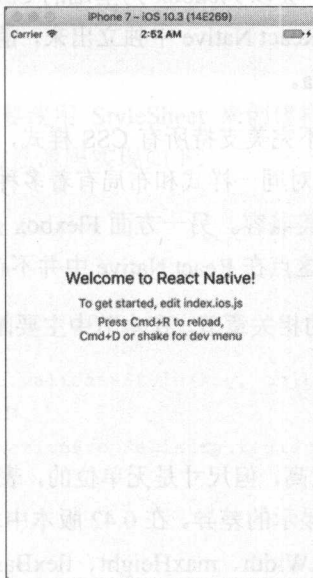
(4) 创建 React Native 项目, 命令如下:

```
react-native init myProject
```

(5) 运行项目，命令如下：

```
cd myProject
react-native run-ios // 或 run-android
```

实例运行效果如图 10.6 所示。



10.6 React Native 运行实例

(6) 打包项目文件，命令如下：

```
react-native bundle --platform android
```

注意：打包出来的是 JSbundle，需要通过原生容器启用，直接打包应用因为涉及签名或开发者账号，读者可前往 React Native 官网了解，地址为 <https://facebook.github.io/react-native/>。

3. React Native 使用

React Native 提供了一系列的组件和 API，调用方式推荐使用 ECMAScript 6 的 import 语法，通过解构的方式，可以减少重复输入，代码如下：

```
import {
  AppRegistry,
  StyleSheet,
  Text,
```



```
View
} from 'react-native'; // 从 React Native 中获得组件和方法，在开发代码中直接使用
```

10.3.2 React Native 样式和布局

React Native 的样式和布局是一套以 Flexbox 为基础的 CSS 子集方案，这套被称为 Yoga（早期叫作 CSSLayout）的方案已经从 React Native 中独立出来，能够在多个语言上开发使用，项目地址为 <https://github.com/facebook/yoga>。

Yoga 是 CSS 的子集意味着并不完美支持所有 CSS 样式，且布局仅能使用 Flexbox 实现，这主要也是因为在 CSS 的发展过程中对同一样式和布局有着多种实现的方式，这对于一个需要转换到原生布局的中间语言来说很难完美兼容。另一方面 Flexbox 是目前最易用的 CSS 布局方式，在浏览器端往往只受限于兼容性，而这点在 React Native 中并不存在。

关于 Flexbox 布局，详见本书的相关章节，在本节中主要阐述 React Native 样式与布局同浏览器中的异同。

1. 高度和宽度

React Native 支持设置固定的宽高，但尺寸是无单位的，表示的是与设备像素密度无关的逻辑像素点，所以在不同的设备上会有显示的差异。在 0.42 版本中增加了对 padding、margin、width、height、minWidth、minHeight、maxWidth、maxHeight、flexBasis 等的百分比支持，方便开发者进行自适应布局。

2. StyleSheet

React Native 的样式属于 JavaScript 对象，要注意使用驼峰命名方式，如果值为字符串需要使用引号。其有两种主流的 CSS 写法，一种是类内联写法，直接将样式放在对象中设置在组件的 style 属性中，或者将多个对象放在一个数组中使用，实例代码如下：

```
<View style = {{
  flex: 1,
  justifyContent: 'center', // 在主轴居中
  alignments: 'center'      // 在交叉轴居中
}}>
</View>
```

类内联写法存在样式无法复用的问题，React Native 提供了 StyleSheet 这个 API，开发者可以在 StyleSheet 中嵌套对象，然后通过使用对象的 Key 来实现样式复用，实例代码如下：

```
const styles = StyleSheet.create({
```

```

container: {                                     // 在 styles 下设置子对象
  borderRadius: 4,
  borderWidth: 0.5,
  borderColor: '#d6d7da'
},
});
<View style = {styles.container}>
</View>

```

不过事实上, 开发者并不一定要使用 `StyleSheet` 来创建样式对象, 普通的对象也可以作为参数传入, `StyleSheet` 对象的 `create` 方法源码实现如下:

```

module.exports = {
  create (obj) {
    const result = {};
    // 遍历所有子对象
    for (var key in obj) {
      StyleSheetValidation.validateStyle(key, obj);
      // 在 register 中注册 ID
      result[key] = ReactNativePropRegistry.register(obj[key]);
    }
    return result;
  }
}

```

`ReactNativePropRegistry` 私有对象的 `register` 方法会生成一个可以通过 ID 来引用的样式表, 并在上下文环境中进行缓存。

3. 样式差异性

在 React Native 中的样式与 CSS 中存在差异, 注意点如下。

- `flex` 只支持整数, `flexGrow`、`flexShrink`、`flexBasis` 与 CSS 一致。
- `flexDirection` 的默认值是 `column`, 在 CSS 中为 `row`。
- `padding` 设置的值同时对 `paddingTop`、`paddingBottom`、`paddingLeft` 和 `paddingRight` 生效, 不支持 “0 3px 3px 0” 这类写法。
- 支持 `paddingHorizontal`, 同时设定 `paddingLeft` 和 `paddingRight`; 支持 `paddingVertical`, 同时设定 `paddingTop` 和 `paddingBottom`。
- `position` 默认为 `relative`, 所以 `absolute` 总是相对于父元素。
- 不支持 `filter` 等 CSS 效果。

10.3.3 React Native 组件概念

React Native 的组件概念来自于 React 框架，组件的本身是一个函数方法，其中的 `render` 函数最终会渲染出一棵虚拟 DOM 树，在 React 中虚拟 DOM 的节点称为 `ReactNode`，大部分的节点最终都会解析成 `TextElement` 或者是 `DomElement`，再通过 `Patch` 将虚拟 DOM 渲染为真实的 DOM。React Native 充分利用了 React 生成虚拟 DOM 的优势，再解析出虚拟 DOM 树，通过调度到原生端并调用原生组件进行渲染。

类比 React，React Native 提供了一套类似于 DOM Node 的组件供开发者使用，在 React 中，所有的 DOM Node 其实也被包装成了组件，再由组件渲染成 DOM 元素供浏览器使用。React Native 中，供原生端使用的 Element 大多会标注为“RCT”开头，通过 `requireNativeComponent` 引入，再由 JavaScript 包装成组件供开发者使用。

如同学习 HTML 开发，开发者需要学习 React Native 的组件使用，React Native 官网提供了详细的组件功能和参数 (Props) 介绍。React Native 组件主要有两类，第一类组件直接对应原生视图，在 JavaScript 代码较少，功能比较基础，在本节中称为简单组件；第二种组件会依赖于第一种组件，通常是第一种组件在 JavaScript 的再封装，在本节中称为复合组件。

10.3.4 简单组件实战

View 是一个典型的简单组件，类似于 HTML 开发中的 DIV 标签。View 是一个容器标签，让开发者能够进行视图布局。与 DIV 不同的是，View 组件内不支持直接在内部添加文本 (字符串)，如果要展示文本，需要使用 Text 组件。同时 View 在子元素的高度超过屏幕高度时也不会默认滚动，在需要滚动的视图里，需要选择 `ScrollView` 或者使用 `onLayout` 进行重新定位。

View 支持的常用 Props 如下。

- `accessible`: Boolean 类型，表示是否启用了无障碍功能的元素。
- `onLayout`: 函数，组件挂载或者布局变化的时候触发。
- `style`: 组件的样式。

这是 View 最常用的几个属性，事实上 View 支持十多种属性，有些是平台限定的，例如 `collapsible`，只能使用在 Android 上，如果一个 View 只用于布局子组件，启用后会被优化移除出原生布局树中。

View 的实例如下：

```
<View style={{borderColor: '#527FE4', borderWidth: 5, padding: 10}}>
```

```
<Text style={{fontSize: 11}}>5px blue border</Text>
</View>
```

效果如图 10.7 所示。

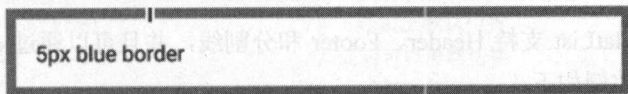


图 10.7 View 组件实例

10.3.5 复合组件实战

React Native 在 0.43 版本中引入了 FlatList 来解决原 ListView 的性能问题，无论是 FlatList 还是 ListView 都属于复合组件，没有对应的原生端组件。FlatList 封装了 VirtualizedList，而 VirtualizedList 则是由 ScrollView 包装了多个 View 组件组成的。ListView 官方使用实例如图 10.8 所示。

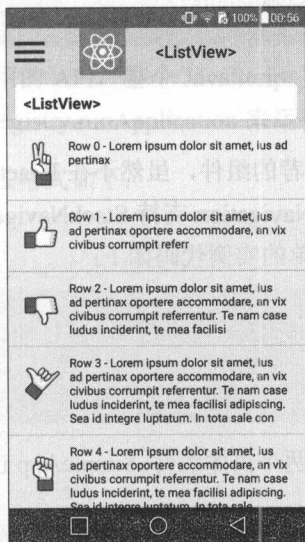


图 10.8 ListView 官方实例

FlatList 这种在 JavaScript 封装简单组件来提供更丰富功能的思想，解决了原生组件跨平台 API 差异性大的问题，也提供了更丰富的功能。

使用 FlatList 相对使用 View 组件要复杂一些，需要定义一个数组数据源，通过 renderItem 来定义每个单元的元素，实例代码如下：


```
<FlatList
  data={[{key: 'a'}, {key: 'b'}]}
  renderItem={({item}) => <Text>{item.key}</Text>} />
/>
```

在复杂场景中, FlatList 支持 Header、Footer 和分割线, 并且可以通过 onRefresh 参数支持下拉刷新, 一个复杂的实例如下:

```
<AnimatedFlatList
  ItemSeparatorComponent={SeparatorComponent}
  ListHeaderComponent={HeaderComponent}
  ListFooterComponent={FooterComponent}           // 分割线
  data={filteredData}                             // 数据
  horizontal={this.state.horizontal}              // 滚动方向
  keyExtractor={this._keyExtractor}               // 生成 key, 优化用
  onRefresh={this._onRefresh}                     // 刷新函数
  ref={this._captureRef}                          // ref 引用
  renderItem={this._renderItemComponent}
/>
```

10.3.6 第三方组件实战

React Navigation 是一个官方推荐的组件, 虽然不在 React Native 的默认代码中, 但是推荐使用其替换内置的 Navigator。React Navigation 支持 StackNavigation 和 TabNavigation, 分别表示堆栈式页面切换和 Tab 切换, 一个简单的实例代码如下:

```
const SimpleApp = TabNavigator ({
  Home: { screen: HomeScreen },
  Chat: { screen: ChatScreen }
});
```

HomeScreen 和 ChatScreen 对应两个页面组件, SimpleApp 也是一个组件, 可以通过 AppRegistry 挂载到应用上。

SimpleApp 也可以作为组件放入 JSX 语法中, 并可以给页面传递额外参数, 代码如下:

```
<SimpleApp
  screenProps={}
/>
```

StackNavigator 和 TabNavigator 支持 StackNavigatorConfig 或 TabNavigatorConfig 作为第二参数, 可以将上面的例子进行扩展, 代码如下:

```
const SimpleApp = TabNavigator ({
  Home: { screen: HomeScreen },
  Chat: { screen: ChatScreen }
}, {
  tabBarPosition: 'bottom',           // tabBar 默认位置
  swipEnabled: true,                  // 是否可以滑动切换
  lazy: true                          // 支持懒加载
});
```

10.3.7 常用 API 实践

在 React Native 中除了组件，还有 API 的概念。组件一般带有 UI 特性，引入方式为在 render 的 JSX 语法中，是一种声明式的调用。API 则是其内部暴露的方法，通过 JavaScript 对象调用，可以用来获得一些原生提供的能力，类似于 Cordova 中插件提供的功能。

在本节会介绍几个常用的 API，更多的 API 可以去 React Native 官方网站学习。

1. AppRegistry

AppRegistry 是开发者必须要使用的 API，整个 JavaScript 需要调用此 API 实现在原生应用处的注册，原生应用通过对应的 AppRegistry.runApplication 来运行程序。

AppRegister 提供以下方法，语法格式如下：

```
// 注册配置
static registerConfig(config:Array<AppConfig>)
// 注册入口组件
static registerComponent(appKey: string, getComponentFunc: ComponentProvider):
// 注册监听函数
static registerRunnable(appKey: string, func: Function)
// 获得在 registerRunnable 中注册的 key
static getAppKeys()
// 运行 App
static runApplication(appKey: string, appParameters: any)
// 从根标签上注销应用组件
static unmountApplicationComponentAtRootTag(rootTag: number)
```

一般来说我们的应用中只需要使用 registerComponent 方法，代码如下：

```
AppRegistry.registerComponent('App', () => App);           // 注册 App 组件到 App 项目中
```

2. AsyncStorage

AsyncStorage 对应着 HTML 5 的 LocalStorage，是一个简单的、异步的、持久化的键值对存储

系统,常用的方法语法格式如下:

```
// 根据键来获得值, 值作为参数返回在回调函数中
static getItem(key: string, callback: (error, result))
// 设置键和值
static setItem(key: string, value: string, callback: (error: Error))
// 移除键和值
static removeItem(key: string, callback: (error: Error))
```

3. Platform

React Native 在 Android 端和 iOS 端可能存在差异,所以有时候开发者会需要进行差异化开发,React Native 提供了 Platform 这个 API 来实现特定平台代码,代码如下:

```
return Platform.OS === 'android' ? executeAndroid(): executeIOS(); // 获取当前平台代码
```

Platform API 除了提供 OS 属性外,还提供了 select 方法来直接运行不同的代码,代码如下:

```
var Component = Platform.select({
  ios: () => require('ComponentIOS'),
  android: () => require('ComponentAndroid')
})
```

4. 数据获取

在 React Native 中,获取数据的方式有 XMLHttpRequest 和 Fetch,笔者推荐使用 Fetch。Fetch 可以使用 Promise 写法或者 Async 的方式,实例代码如下:

```
// Promise 写法
getData() {
  return fetch(url)
    .then(response => response.json()) // 将数据转化为 json 对象
    .then(responseJson => responseJson.data)
    .catch(err => console.log(err))
}
// Async 写法
async getData() { // async 函数
  try { // 使用 try catch 捕获异常
    let response = await fetch(url); // 在 await 处暂停等异步结束
    let responseJson = await response.json();
    return responseJson.data;
  } catch(err) {
    console.log(err);
  }
}
```


这里通过一个前端开发中常会遇到的“小红点”提示的例子来展示 AsyncStorage 和 Fetch 一起使用的场景。假设有一个显示信息的栏位，如果获取了新信息则显示小红点，当用户单击小红点后消失，代码如下：

判断是否有红点数据，获取信息代码如下：

```

async _setLinkStatus() {
  let data;
  try {
    // 链接状态数据，key 为 url，value 为是否显示红点
    data = await AsyncStorage.getItem('linkStatus');
  } catch(err) {
    console.error(err);
  }
  let response, responseData, responseValue;
  try {
    response = await fetch(url); // url 为请求地址
    responseData = await response.json();
    responseValue = responseData.data;
  } catch(err) {
    console.error(err)
  }
  data = data? JSON.parse(data) : {}; // 把键值对象转化为 json 对象
  if (responseData && responseData.status === 0) { // 如果成功获取数据
    let linkStatus= {}; // 一个空对象存当前状态
    for (let i = 0; i < responseValue.length; i++ ) { // 遍历数据
      let { url } = responseValue[i]; // 取出 url
      if (Object.keys(data).includes(url)) { // 如果包含这条 url
        linkStatus[url] = data[url]; // 同步状态到 linkStatus
      }
      linkStatus[url] = true; // 没有说明是新链接，未单击过
    }
    this.setState({linkList: linkList}); // 设置当前组件的 state
    if (linkStatus!== data) { // 如果有新链接，则覆盖
      AsyncStorage.setItem('linkStatus', JSON.stringify(linkStatus));
    }
  }
}

```


单击后清除红点状态, 代码如下:

```
_onPressButton(value, key) {
  if (value.isNew === false) // 如果没单击过
    let linkStatus = deepClone(this.state.linkStatus); // 深拷贝一份当前状态
    linkStatus[key] = false; // 单击过设置为 false
    AsyncStorage.setItem('linkStatus', JSON.stringify(linkStatus))
      .then(() => this._redirectUrl(value.url)); // 在异步回调中跳走
  } else {
    this._redirectUrl(value.url); // 单击过直接跳走
  }
}
```

10.4 实战演练: 用 React Native 开发新闻阅读应用

本节将通过运用 React Native 开发一个新闻阅读应用, 实例中提供了基本的新闻阅读功能, 包括 TabBar 切换、导航切换、新闻列表、评论查看, 以及打开 WebView 查看具体新闻内容等功能。

使用 React Native 开发应用的体验接近 React, 会存在大量 React 代码, 如果对 React 不熟悉的读者可以先阅读相关章节。在本项目中选择的新闻源为 HackerNews。

提示: HackerNews 是一个适合程序员了解业界新闻的网站, 其提供了公开的 API 供开发者使用, 官网地址为 <https://news.ycombinator.com/>。

10.4.1 React Native 的工程项目结构一览

React Native 的工程项目目录一般会包含 android 和 ios 文件夹用来存放原生端工程代码, 入口文件 Android 端为 “index.android.js”, iOS 端为 “index.ios.js”, 项目的 JavaScript 代码一般存放在 src 文件中, 工程结构如图 10.9 所示。

在这个项目中, containers 存放页面, components 存放通用组件, services 存放公共方法和常量。因为项目并没有在端入口作差异化代码, 所以在 index.ios.js 和 index.android.js 中代码如下:

```
import React, { Component } from 'react';
import { AppRegistry } from 'react-native';
import App from './src/main'; // main 文件中存放 JS 入口
AppRegistry.registerComponent('ReactNativeHW', () => App); // 注册到 App 中
```



图 10.9 React Native 工程实例

10.4.2 列表页

在 main 文件中存放 Tab 的导航切换，导航组件选用 react-navigation，代码如下：

```
const App = TabNavigator({                                // route 配置
  Ask: { screen: Story, navigationOptions: { tabBarLabel: 'Ask HN' } },
  Show: { screen: Story, navigationOptions: { tabBarLabel: 'Show HN' } },
  Top: { screen: Story, navigationOptions: { tabBarLabel: 'Top Stories' } },
  New: { screen: Story, navigationOptions: { tabBarLabel: 'New Stories' } },
  Job: { screen: Story, navigationOptions: { tabBarLabel: 'Jobs' } }
}, {                                                       // tab 配置
  tabBarPosition: 'bottom',                                // tabBar 位置
  swipeEnabled: true,                                     // 允许滑动切换
  lazy: true,                                              // 懒加载
  initialRouteName: 'Top',                                // 默认开始路由
  tabBarOptions: {                                         // tabBar 配置
    activeTintColor: '#FF6600',                            // 选中的 tab 的文字和图标颜色
    labelStyle: { marginBottom: 16 }                        // label 样式
  }
});
```

本项目中存在导航嵌套，外层为 Tab 导航，内层是 Stack 导航，在 Story 组件中配置内层导航，代码如下：

```
const Story = StackNavigator({
  // StackNavigator 的路由
  { List: { screen: StoryList }, Detail: { screen: StoryDetail }, View: { screen: StoryView } },
  { navigationOptions: {
    headerStyle: { backgroundColor: '#FF6600'},
    headerTintColor: '#FFFFFF'
  }
}
);
// 输出在 JSX 中可以传递额外的参数 screenProps
export default props => <Story screenProps={props.navigation.state} />;
```

StoryList 实现了整个新闻列表页, HackerNews 提供的 API 能够拿到每个栏目的新闻 ID 数组, 开发者需要通过这个数组调用详情接口获得新闻的标题等内容。

注意: ID 数组会长达 500 条, 如果一次性访问 500 次详情接口会导致应用长时间处于未响应状态, 另外响应的结果应该等待数个接口获得数据后统一写入 State 中再触发列表渲染, 如果每条数据都直接写入 State, 会导致列表频繁重渲染, 影响用户体验。

在本项目中, 选择了分页获取的方法, 第一次会拉取 10 条数据, 当所有数据请求完毕后再统一写入 State, 利用 FlatList 组件提供的 onEndReached 方法, 在用户滚动到页面底部时拉取后 10 条数据, 实现代码如下:

```
_getList = () => {
  // 获取页面名称调用相关 API 获得新闻 id 数组
  const { screenProps } = this.props;
  fetch(Api['HN_${screenProps.routeName.toUpperCase()}_STORIES_ENDPOINT'])
    .then(response => response.json())
    .then(responseJson => {
      // 设置 state 并调用获得具体数据接口
      this.setState({ listId: [...responseJson], pageNum: 1 }, () =>
        { this._getIdInfo(true); });
    })
    .catch(err => { console.log(err); });
}

_getIdInfo = (refresh = false) => {
  // 是否需要刷新
  let { pageNum, listId, listInfo, loading } = this.state;
  // 如果在 loading 或者数据全部拉取完毕这返回
  if (loading || (!refresh && pageNum * NUMBER_PER_PAGE > listId.length)) return;
```

```

// 设置 loading 在回调中调用接口
this.setState({ loading: true }, () =>
  this._setListInfo({ pageNum, listId, listInfo, refresh })
);
}

_setListInfo = ({ pageNum, listId, listInfo, refresh }) => {
  // 将 10 条数据拉取封装在 promise 中
  let promises = fetchLimitData({
    pageNum, numPerPage: NUMBER_PER_PAGE, source: listId });
  // 用 Promise.all 在 10 条全部拉取完后再执行回调函数
  Promise.all(promises) .then(infos => {
    this.setState({
      // 如果是刷新, 直接把数据覆盖原来的数据, 不然则合并到原来数据上
      listInfo: refresh ? infos : listInfo.concat(infos),
      pageNum: pageNum + 1,           // 下次拉取下一页
      loading: false                 // 关闭 loading 状态
    });
  }).catch(err => { this.setState({ loading: false }); console.log(err); });
}

```

因为获取部分数据的方法会在评论里复用, 这里选择抽出到公共方法中, 代码如下:

```

export const fetchLimitData = ({ pageNum, numPerPage, source }) => {
  // 截取部分列表 id, map 后调用数据
  return source.slice((pageNum - 1) * numPerPage, pageNum * numPerPage)
    .map(id => {
      return fetch(Api.HN_ITEM_ENDPOINT + id + '.json').then(response =>
        response.json()
      );
    });
};

```

因为 FlatList 也会被复用, 被封装在 RefreshList 组件中调用, JSX 代码如下:

```

<FlatList
  ListHeaderComponent={props.renderListHeader}           // 渲染头部组件
  keyExtractor={this._keyExtractor}                       // FlatList 每个 item 要有 key
  data={props.data}
  onRefresh={props.onRefresh}
  refreshing={false}
  onEndReached={props.onEndReached}
  renderItem={props.renderItem}
/>

```


除了滚动加载，这个页面也支持下拉刷新，只需要将 `getList` 方法作为 `FlatList` 的 `onRefresh` 参数传入即可以实现，至此整个列表页基本完成，效果如图 10.10 所示。

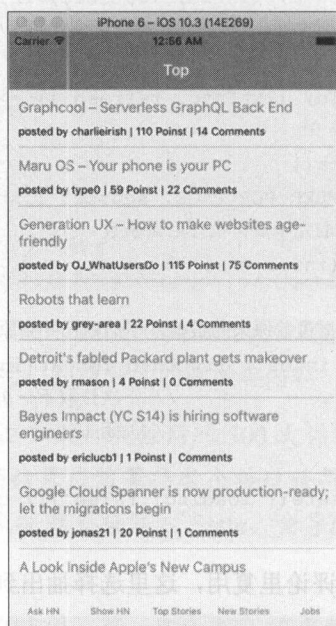


图 10.10 新闻列表页面

10.4.3 新闻评论页

新闻评论页的实现跟列表页类似，利用 `FlatList` 显示评论，不同点在于，数据并不完全依赖接口，可以将列表页拿到的数据作为参数传给评论页面，代码如下：

```
<TouchableOpacity onPress={() => navigate('Detail', { ...data })}> // 传递 data 参数
...
</TouchableOpacity>
```

在评论页面可以通过 `props.navigation.state.params` 获取数据，评论页的具体实现请参见项目源码，评论页效果如图 10.11 所示。

10.4.4 新闻展示页

最后是展示新闻内容的 `WebView` 页面，`React Native` 提供了 `WebView` 组件来显示页面，使用方法如下：

```
render() {
  const { params } = this.props.navigation.state;
  return <WebView source={{ uri: params.url }} />; React Native 提供的 WebView 组件
}
```

最终的 WebView 页面效果如图 10.12 所示。

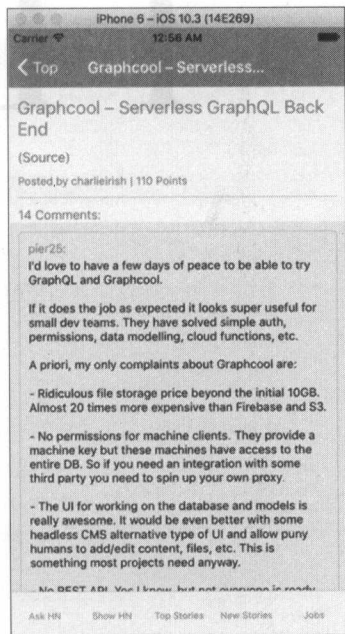


图 10.11 新闻评论页面



图 10.12 新闻内容页面

至此，一个 React Native 实现的新闻阅读应用就完成了，一些具体的实现没有全部在本节中讲述，读者可以阅读本章源码来了解，并继续优化或者扩展本应用。

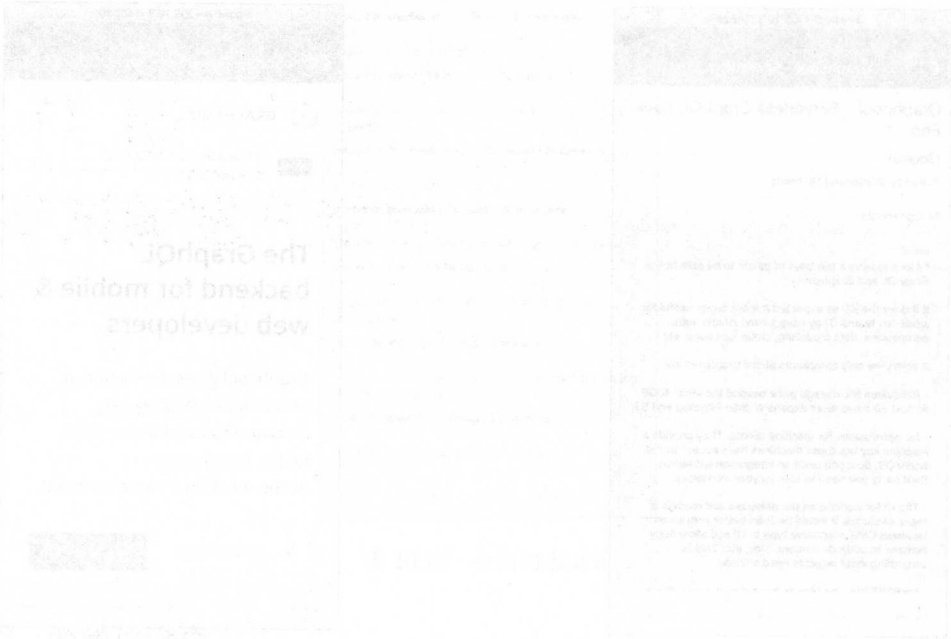
10.5 本章小结

本章主要介绍了混合式开发的相关知识，简述了 Cordova 和 React Native 这两种不同的开发工具。Cordova 类似传统的 Web 开发，但是性能较差，React Native 虽然性能较好，但有一定的学习成本，在实际项目中，根据需求灵活选择。

需要注意的是，React Native 仍处在 Beta 版本，在正式版本前，API 都可能有很大的变化。Cordova 相对稳定，但在一些第三方插件选择上，可能会出现编译问题，开发者在排查问题时，需

要留意插件是否被当前 SDK 版本兼容。

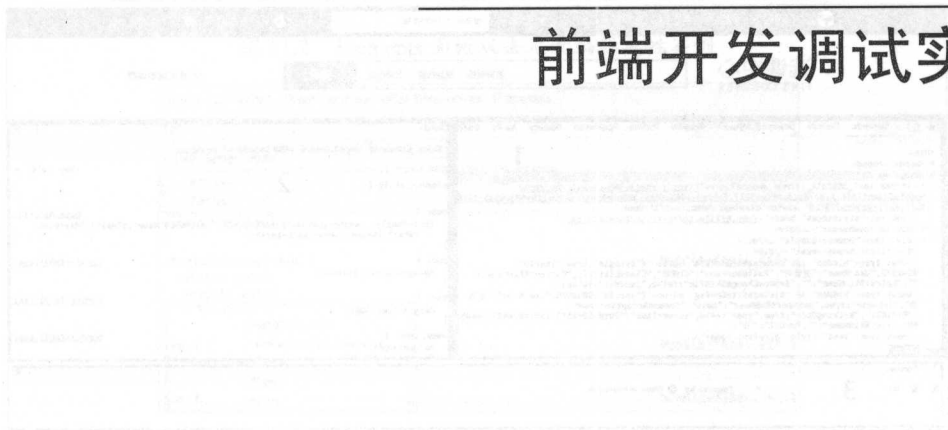
现在的混合式开发已经在 Web 端或 JavaScript 中提供了很好的封装，开发者在完全不熟悉原生代码的情况下也能够开发出独立的应用。但是想要深入混合式开发，学习原生代码和 API 仍是最好的途径。



11

第 11 章

前端开发调试实战



开发调试是必须学会的技能，其重要性不言而喻，本章将从浏览器调试、代理工具、多终端同步工具、模拟器调试、多平台调试、云真机调试和 React 调试这 7 个方面介绍实际开发过程中调试工具的使用。读者通过本章的学习，可以掌握常用的调试手段，从而降低开发成本，提高工作效率。

11.1 浏览器调试

Web 应用的开发，离不开浏览器调试。浏览器调试能够实时查看页面效果，修改结果并立即生效，也能够查看页面的报错和断点调试错误，甚至能查看页面的性能从而有针对地对页面进行优化。诸多好处都依赖于强大的浏览器开发者工具，本章主要介绍 Chrome 和 Safari 两款现代浏览器的开发者工具，帮助大家在日常开发中熟练地使用调试方法。

11.1.1 Chrome 开发者工具

Chrome 非常强大,是最流行的浏览器之一,相信大家都使用过这款浏览器。如果没有安装 Chrome,请前往官网 <http://www.google.cn/chrome/browser/desktop/index.html> 下载。本节主要介绍 Chrome 浏览器的开发者工具。

首先来认识下 Chrome 开发者工具界面。打开 Chrome,访问 <http://class.hujiang.com>,单击鼠标右键,选择“检查”或者“审查元素”打开 Chrome 开发者工具界面。也可以使用快捷键,如果是 Windows 系统可以键入“F12”,如果是 Mac OS 系统可以使用“Command + Option + I”组合键来打开 Chrome 开发者工具。界面如图 11.1 所示。



图 11.1 Chrome 开发者工具界面

从图 11.1 中看出,默认的 Chrome 开发者工具界面分成了三个部分。第一部分是整个页面 HTML 结构;第二部分是页面当前选中元素的 CSS;第三部分是当前页面的 Console 控制台。Chrome 开发者工具面板的顶部有一排 Tab 标签,当前选中的是 Elements 面板(其他还包含有 Console、Sources、Network、Timeline、Profiles、Application、Security、Audits 等面板)。图 11.1 中还有最后一个 EditThisCookie 面板,这是 Chrome 浏览器的扩展程序,非默认安装,开发者可以前往 Chrome 网上应用商店获取,地址为 <https://chrome.google.com/webstore/category/extensions?hl=zh-CN>。

1. Elements 面板

Elements 面板主要用来实时查看、编辑页面 DOM 结构和 CSS 样式。双击 DOM 树视图中的节点,可以实时编辑标签属性,修改的效果会立刻反映在浏览器页面,如图 11.2 所示。

右侧的样式面板可以对元素的 CSS 属性和值进行实时修改,也可以为元素添加新的 CSS 样式和设置元素 active、hover、focus、visited 四种状态,如图 11.3 所示。



图 11.2 Elements 面板双击元素编辑属性实例

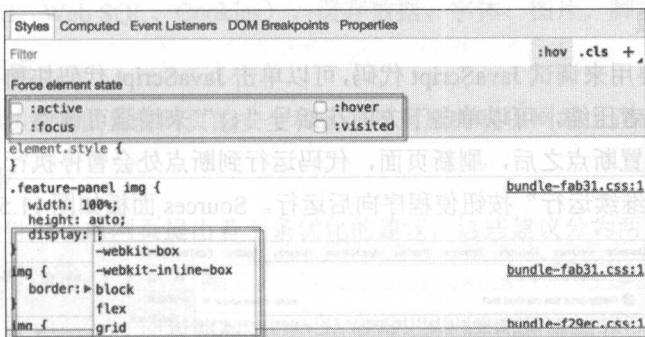


图 11.3 样式面板修改 CSS 实例

单击右侧的 Computed 面板，可以查看元素所有计算得出的样式，可以编辑左侧选中的盒子模型数据。单击不同的位置（top、bottom、left、right）可修改元素的 padding、border、margin 属性值，如图 11.4 所示。

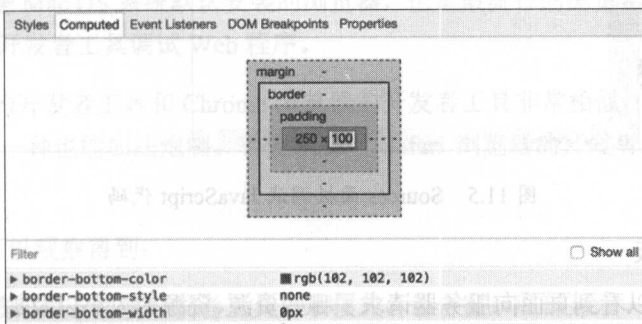


图 11.4 Computed 面板

图 11.4 的标签选择栏中, 其他面板说明如下。

- Event Listeners: 用于显示当前 DOM 节点和其祖先节点的所有 JavaScript 事件监听器。
- DOM Breakpoints: 用于列出所有的 DOM 断点。
- Properties: 全面地列出当前选中内容的属性。

2. Console 面板

Console 面板用于显示控制台输出日志, JavaScript 代码中使用 `console.log`、`console.warn` 和 `console.error` 方法将日志信息输出到控制台。这是调试 JavaScript 代码常用的方法, 可以在关键代码位置打印出变量的值, 从而获取当前运行环境的上下文内容。Console 面板还有一个非常常用的功能, 输入 JavaScript 表达式来实时计算编码。

3. Sources 面板

Sources 面板主要用来调试 JavaScript 代码, 可以单击 JavaScript 代码块前面的数字设置断点。如果当前代码被开发者压缩, 可以单击下方的花括号 “{}” 来增强可读性, 所有的断点都会列在右侧的断点区。设置断点之后, 刷新页面, 代码运行到断点处会暂停执行, 通过右侧面板上的“步进”按钮或者“继续运行”按钮使程序向后运行。Sources 面板如图 11.5 所示。

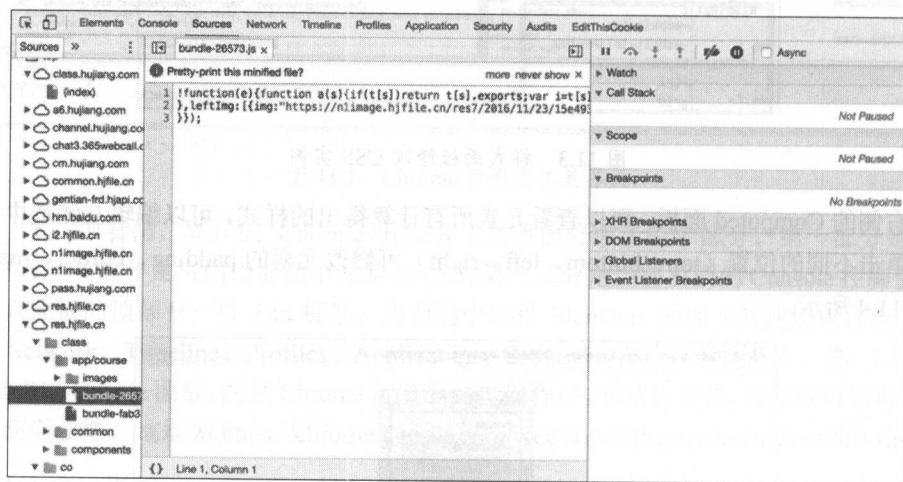


图 11.5 Sources 面板调试 JavaScript 代码

4. Network 面板

Network 面板可以看到页面向服务器请求了哪些资源、资源的大小以及加载资源花费的时间, 当然也能看到哪些资源不能成功加载。此外, 还可以查看 HTTP 请求的详细信息、响应内容等。

5. Timeline 面板

Timeline 面板可以用来分析网页的性能状况,该面板功能十分强大,可以计算包括处理 DOM 事件、页面布局渲染或者向屏幕绘制元素所花费的时间。Timeline 也可以通过事件、框架和实时内存用量 3 个方面的数据来监测网页。通过这些数据,开发者可以方便地找出页面中存在的问题。

6. Profiles 面板

Profiles 面板是对 Timeline 面板信息的补充,也可以称为 Timeline 的数字化版本,利用该面板可以追踪网页程序的内存泄漏问题,帮助开发者进行判断从而进一步提升 JavaScript 执行性能。

7. Application 面板

Application 面板主要是记录网站加载的所有资源信息,包括存储数据(Local Storage、Session Storage、IndexedDB、Web SQL、Cookies)、缓存数据、字体、图片、脚本、样式表等。

8. Security 面板

通过该面板可以去调试当前网页的安全和认证等问题并确保网站上正确地实现 HTTPS。

9. Audits 面板

Audits 面板会针对目前网页提出若干条优化的建议,这些建议分为两大类,一类是网络加载性能,另一类是界面性能。

通常,可以使用 Elements 面板修改 DOM 结构和 CSS 样式来调试界面。使用 Console 面板、Network 面板、Sources 面板、Application 面板配合调试 JavaScript 代码和查看 AJAX 请求,使用其余的面板来优化网络和程序性能。

11.1.2 Safari 开发者工具

Safari 浏览器是 Mac OS 系统默认安装的浏览器,也是最流行的浏览器之一。本节主要介绍利用 Safari 浏览器的开发者工具调试 Web 程序。

Safari 浏览器的开发者工具和 Chrome 浏览器的开发者工具非常相似,掌握了其中一种浏览器的调试方法,另外一种也能如法炮制。首先来看下 Safari 浏览器的开发者工具的界面,如图 11.6 所示。

从图 11.6 中可以观察得到:

- 区域 1 的按钮用于切换开发者工具显示模式,包含窗口底部显示、窗口右侧显示和独立于窗口之外显示。

- 区域 2 的按钮用于刷新页面和下载页面。
- 区域 3 是整个网站运行状况概览。
- 区域 4 的按钮是用来选择元素的,也是使用频率非常高的按钮。
- 区域 5 是搜索框,可以搜索 HTML、CSS、JavaScript。

上述区域的下方有 7 个切换标签,和 Chrome 浏览器的开发者工具类似,用于切换面板,包含元素、网络、资源、时间线、调试器、存储空间和控制台。



图 11.6 Safari 浏览器的开发者工具界面

1. 元素面板

元素面板和 Chrome 浏览器开发者工具的 Elements 面板类似,主要用来实时查看、编辑页面 DOM 结构和 CSS 样式,双击 DOM 树视图中的节点,可以实时编辑标签属性,修改效果并且实时反映。

2. 网络面板

网络面板和 Chrome 浏览器开发者工具的 Network 面板类似,可以看到页面向服务器请求了哪些资源、资源的大小以及加载资源花费的时间,当然也能看到哪些资源不能成功加载。此外,还可以查看 HTTP 的请求头、返回内容等。

3. 资源面板

资源面板和 Chrome 浏览器开发者工具的 Sources 面板类似,可以查看 HTML、CSS、JavaScript

等资源文件。

4. 时间线面板

时间线面板和 Chrome 浏览器开发者工具的 Timeline 面板类似，主要用于检测网页的性能，无论是网络请求的问题，还是页面渲染的问题，又或是 JavaScript 脚本的问题，都能从时间线面板中窥伺一二。

5. 调试器面板

调试器面板和资源面板配合使用，用于调试 JavaScript 程序。

6. 存储空间面板

存储空间面板和 Chrome 浏览器开发者工具的 Application 面板类似，主要是记录网站加载的所有资源信息，包括本地存储空间、会话存储空间、数据库、已索引的数据库、应用程序缓存、Cookie。

7. 控制台面板

控制台面板和 Chrome 浏览器开发者工具的 Console 面板类似。

Safari 浏览器的开发者工具还有一个非常有用的功能，就是响应式设计模式。通过该模式可以模拟出各种类型的移动端尺寸。在 Mac OS 系统上使用“option+command+R”组合键来开启和关闭响应式设计模式。响应式设计模式的界面如图 11.7 所示。

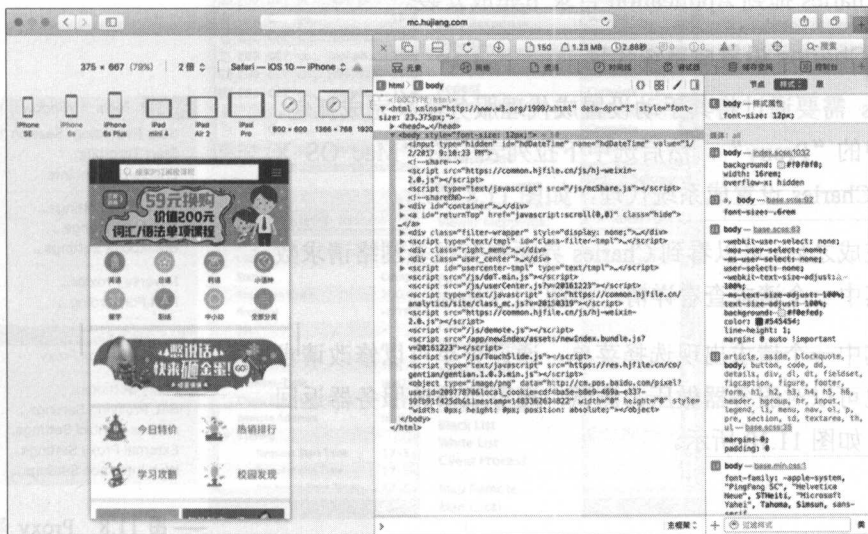


图 11.7 响应式设计模式的界面

从图 11.7 可以看出,通过单击页面上方的设备,可以切换页面显示模式,从而模拟页面在各种设备上的运行状况。

11.2 代理工具

HTTP 代理工具是移动端开发必不可少的工具。通过代理工具可以抓包查看网络请求的具体细节。也可以将线上的文件代理成本地文件,不用重复上线就能调试线上问题。用好代理工具,能够让开发工作事半功倍。这里主要介绍两种常用的代理工具,Mac OS 下的 Charles 和 Windows 下的 Fiddler。

11.2.1 Mac OS 下 Charles 的用法

Charles 是 Mac OS 下常用的网络抓包工具,通过将自己设置成系统的网络访问代理服务器,使得所有的网络访问请求都通过 Charles 来完成,从而实现了网络封包的截取和分析。Charles 是收费软件,有 30 天的试用期,试用期过后也可以继续使用,但是每次使用不能超过 30 分钟。笔者认为,为好工具做一点投资还是值得的。

1. Charles 的安装

安装 Charles,前往 Charles 的官方网站,地址为 <http://www.charlesproxy.com>,文件后缀名“dmg”。打开后将 Charles 拖到 Application 目录下完成安装。

2. 将 Charles 设置成系统代理

Charles 需要通过将其手动设置成代理服务器来完成抓包,选择菜单中的“Proxy”,然后选中下拉列表中的“Mac OS X Proxy”将 Charles 设置成系统代理,如图 11.8 所示。

设置完成之后,可以看到 Charles 界面中有很多网络请求数据,选中其中一个请求查看详情,如图 11.9 所示。

右击其中一个请求出现选择菜单,通过菜单可以修改请求的内容,也可以对服务器做压力测试,还可以修改服务器返回的内容等,如图 11.10 所示。

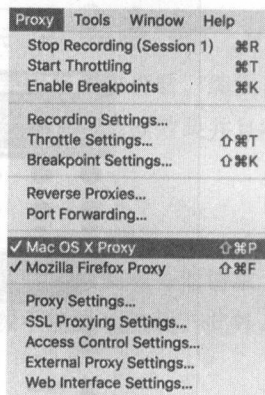


图 11.8 Proxy 菜单

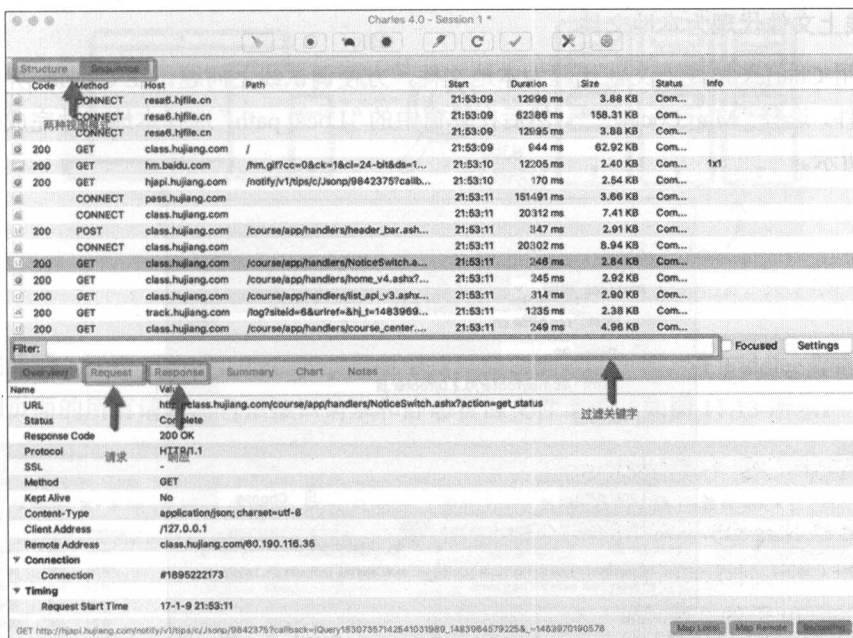


图 11.9 Charles 界面实例

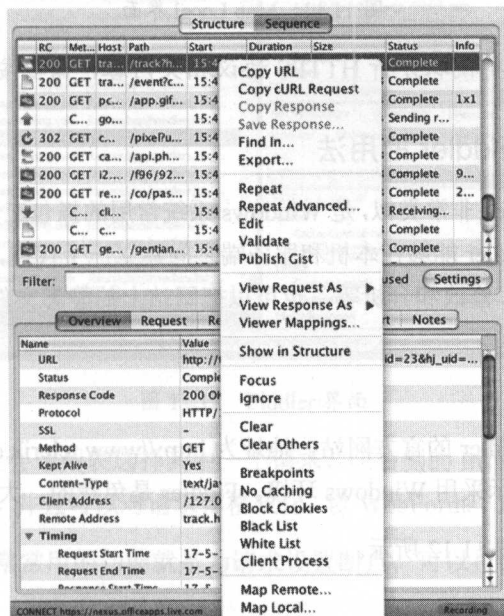


图 11.10 右击请求出现的菜单

3. 把线上文件代理为本地文件

可以使用 Charles 把线上文件代理为本地文件，方便调试线上问题。在 Charles 界面中右击需要代理的文件，选择“Map Local...”，然后在弹窗中的“Local path”选择本地文件完成代理设置，如图 11.11 所示。

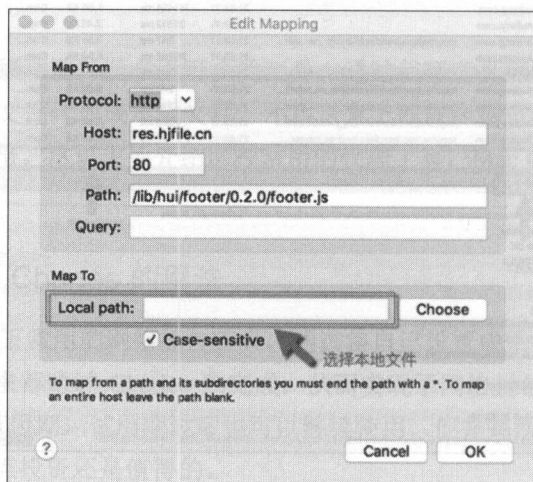


图 11.11 Map Local 界面

注意：开发者如果需要截取分析 HTTPS 协议相关内容，需要安装 Charles 的 CA 证书。

11.2.2 Windows 下 Fiddler 的用法

Fiddler 的用法和 Charles 非常类似，是 Windows 系统常用的抓包工具。当然，现在也有 Mac OS 的 Beta 版本可以使用。Fiddler 能够在本机和服务端之间建立一个代理，通过这个代理，对所有经过的请求和响应进行拦截、修改和分析等，也可以把网站上的静态文件代理为本地的文件，简化调试的流程。

1. Fiddler 的安装

安装 Fiddler，前往 Fiddler 的官方网站，地址为 <http://www.telerik.com/fiddler>，下载最新版本的 Fiddler 安装包，本节演示采用 Windows 环境。Fiddler 是免费的，大家可以放心使用。

Fiddler 的工作原理如图 11.12 所示。

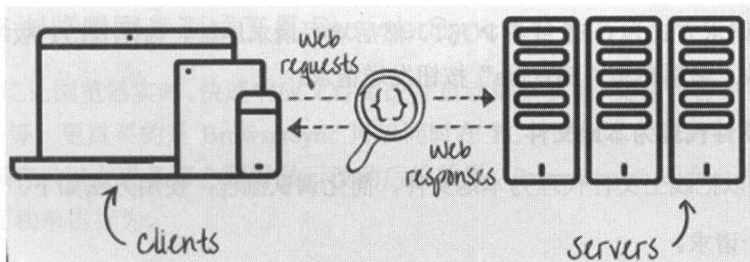


图 11.12 Fiddler 的工作原理

安装完成后，打开 Fiddler，然后在浏览器中打开需要调试的页面，在 Fiddler 界面的会话列表中可以看到页面的所有请求，包括接口请求和静态资源文件请求，如图 11.13 所示。

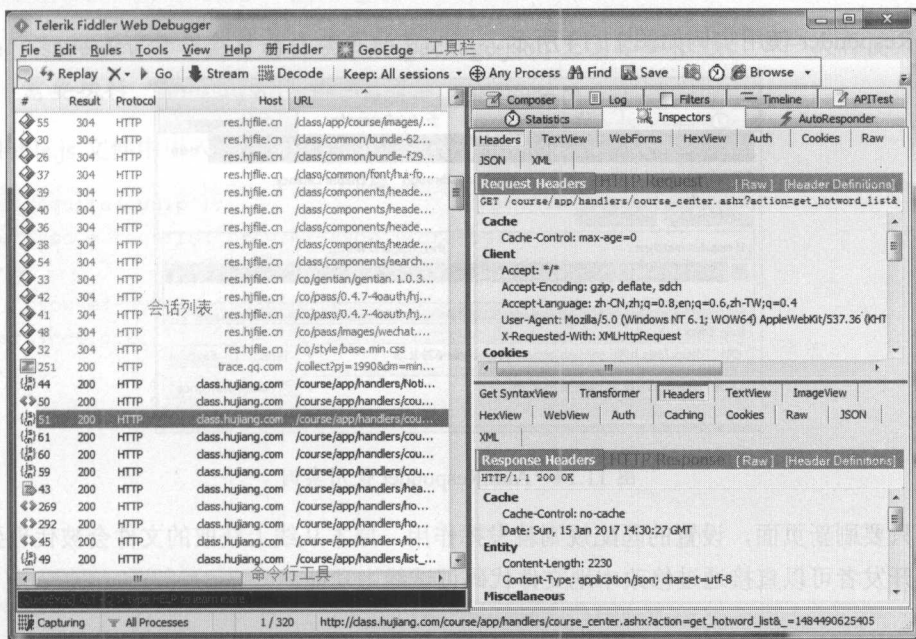


图 11.13 Fiddler 界面

在图 11.13 中左侧面板会话列表选择其中一个请求，在右侧的 Inspectors 窗口中可以详细查看请求的 Headers 和 Cookies，并且可以对请求体格式化，以 WebForms、JSON 和 Raw 等方式查看。

Fiddler 中另外一个非常常用的功能就是为请求设置断点，可以拦截所有匹配的请求，并且为这些请求模拟返回值。

右侧的 Composer 窗口可以模拟发送请求，可以利用这个功能对接口进行测试。使用方法非常

简单, 首先选择请求方式为 GET 或者 POST, 然后填写请求地址, 选择 HTTP 版本, 并在 Request Body 中填写参数, 最后单击“Execute”按钮发送请求。

2. 把线上文件代理为本地文件

Fiddler 也可以把线上文件代理为本地文件, 简化调试流程, 使用方法如下。

- 选中一个请求。
- 在右侧选择“AutoResponder”窗口。
- 勾选“Enable rules”和“Unmatched request passthrough”。
- 单击“Add Rule”按钮。
- 在最下方的文本框中选择“Find a file”, 从本地选择需要代理的文件。

AutoResponder 使用实例如图 11.14 所示。

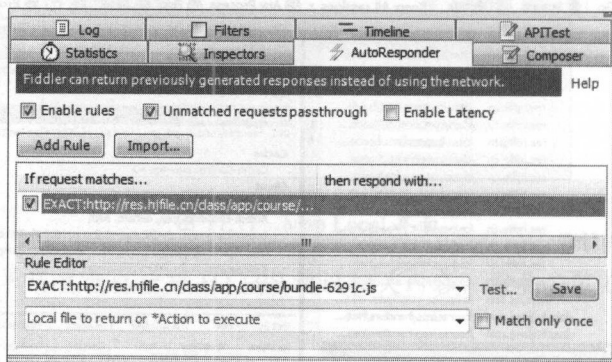


图 11.14 AutoResponder 使用实例

最后只要刷新页面, 设置的匹配规则就发挥作用, 原本从线上获取的文件会被代理到本地文件地址, 开发者可以直接通过修改本地文件代码调试线上功能。

11.3 多终端同步工具

在日常开发中, 常常需要在多个终端中调试页面, 包括 HTML、CSS、JavaScript 等。但是一般的调试方法非常烦琐, 因为每次修改代码之后不能实时看到效果, 每个终端都需要手动刷新查看。调试的改动通常较为频繁, 次数多导致整体的调试效率下降。读者可以利用多终端同步工具来解决这个问题, 本节主要介绍两个比较常用的多终端同步工具 BrowserSync 和 Emmet LiveStyle。

11.3.1 多设备浏览器同步测试工具 BrowserSync

Browsersync 能让浏览器实时、快速响应文件更改并自动刷新页面,文件包含 HTML、JavaScript、CSS、Sass、Less 等。更重要的是 Browsersync 可以同时 PC、平板、手机等设备下进行调试,任何一次代码的保存,以上设备都会同时显示改动后的效果。Browsersync 还有一个可视化界面来控制同步滚动页面和单击行为。

1. 安装

Browsersync 通常和构建工具 Gulp 或者 Grunt 搭配使用,此处使用 Browsersync 和 Gulp 作为演示实例。首先,需要在项目中安装 Browsersync 和 Gulp 依赖包,命令如下:

```
npm install browser-sync gulp --save-dev
```

注意: --save-dev 命令会自动在项目的 package.json 里面添加依赖,下一次再安装时,只需要执行“npm install”即可。

在 gulpfile.js 文件中引入 BrowserSync 包,代码如下:

```
var gulp = require('gulp');
var browserSync = require('browser-sync').create();
// 静态服务器
gulp.task('browser-sync', function() {
  browserSync.init({
    server: {
      baseDir: "./"
    }
  });
});
// 代理
gulp.task('browser-sync', function() {
  browserSync.init({
    proxy: "你的域名或 IP"
  });
});
```

注意: browserSync.init 方法中的 server 参数会创建一个小型静态服务器,可在浏览器中访问 <http://localhost:3000> 对页面进行调试。如果已经存在一个本地的服务器(如本地的 PHP 服务器)进行调试,则可以使用 browserSync.init 方法中的 proxy 参数来启用代理模式。

2. Sass 和 CSS 注入

通过流的方式创建任务流程, 在任务完成后调用 `reload` 方法, 所有的浏览器将被告知变化并实时更新, 实例代码如下:

```
// gulpfile.js 文件
var gulp = require('gulp');
var browserSync = require('browser-sync').create();
var sass = require('gulp-sass');
var reload = browserSync.reload;
var src = {
  scss: 'app/scss/*.scss',
  css: 'app/css',
  html: 'app/*.html'
};
// 静态服务器, 监听 sass 文件和 html 文件变化
gulp.task('serve', ['sass'], function() {
  browserSync.init({
    server: './app'
  });
  gulp.watch(src.scss, ['sass']);
  gulp.watch(src.html).on('change', reload);
});
// 把 sass 文件编译成 css 文件
gulp.task('sass', function() {
  return gulp.src(src.scss)
    .pipe(sass())
    .pipe(gulp.dest(src.css))
    .pipe(reload({stream: true}));
});
gulp.task('default', ['serve']);
```

来看下实例项目的目录结构, 如图 11.15 所示。在浏览器中打开地址 `http://localhost:3000`, 如图 11.16 所示。接下来在 `main.scss` 中修改字体颜色为红色, 保存后无须刷新页面, 页面同步发生改变, 如图 11.17 所示。

从实例项目中可以看出, 修改字体颜色后, 不需要刷新页面就能看到修改后的效果, 如果有多个浏览器打开了该页面, 所有页面均同步进行更新。

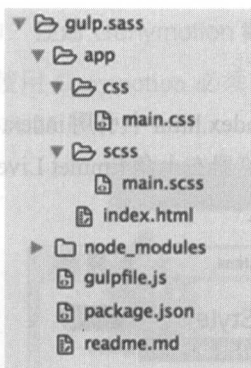


图 11.15 项目目录结构

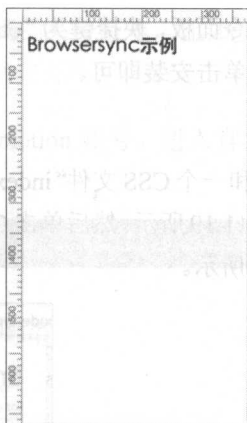


图 11.16 项目实例页面



图 11.17 项目实例页面

11.3.2 双向自动刷新样式工具 Emmet LiveStyle

提到 Emmet 大家应该都不会陌生，该工具可以帮助开发人员高效和快速编写 HTML 和 CSS 的插件。该公司还提供了一款名为 Emmet LiveStyle 的样式实时预览 Chrome 插件，方便开发者调试页面样式。

1. 安装

安装浏览器扩展程序，可以在 Chrome 网上应用商店中搜索“Emmet LiveStyle”，然后单击“添加至 Chrome”完成安装，如图 11.18 所示。



图 11.18 Chrome 扩展程序

安装 Sublime Text 插件，打开其命令面板，快捷键为“ctrl (command)+shift+p”，输入“install package”并搜索关键词“lifestyle”，单击安装即可。

2. 使用

新建一个 HTML 文件“index.html”和一个 CSS 文件“index.css”。在 index.html 中引用 index.css，然后在 Chrome 中打开 index.html，如图 11.19 所示。然后单击 Chrome 浏览器右上角 Emmet LiveStyle 扩展程序按钮，打开开关，如图 11.20 所示。

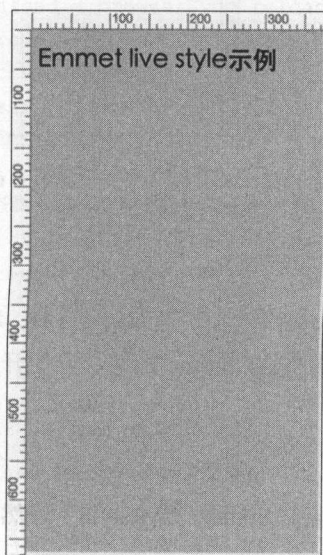


图 11.19 实例页面

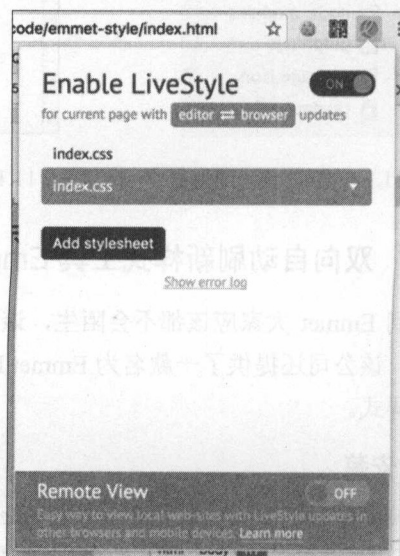


图 11.20 Emmet LiveStyle 扩展程序

接下来在 Sublime Text 中修改页面的样式，不需要刷新页面就能看到修改后的效果。如果在 Chrome 中修改页面的样式，也会反映到 Sublime Text 编辑中。

11.4 模拟器调试

模拟器是运行在本地电脑上的虚拟设备，有效地缓解了开发人员无法获取大量物理设备的难题。本节将介绍目前常用的 Android 和 iOS 设备模拟器。

11.4.1 Android 模拟器调试

比较常用的 Android 模拟器软件推荐 Genymotion。Genymotion 可以选择不同的手机设备，比

如三星、摩托罗拉等，也可以选择不同的系统版本，比如安卓 5.0.0 或者 6.0.0 等。

1. 注册 Genymotion 账号

使用 Genymotion 必须注册 Genymotion 账号。进入官网 <http://www.genymotion.com>，注册页如图 11.21 所示。

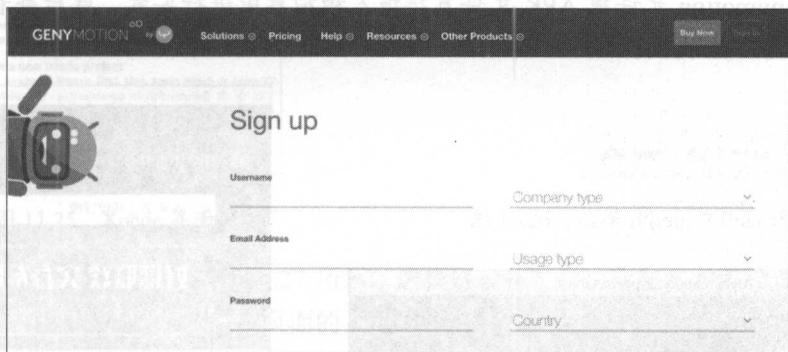


图 11.21 注册 Genymotion 账号

2. 安装 VirtualBox

使用 Genymotion 之前需要安装 VirtualBox，Genymotion 依赖于虚拟环境运行。VirtualBox 可以去官网 <https://www.virtualbox.org/wiki/Downloads> 下载安装。VirtualBox 界面如图 11.22 所示。

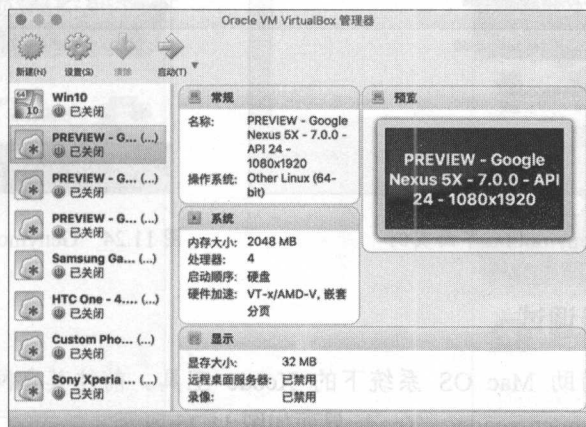


图 11.22 VirtualBox 界面

3. 使用 Genymotion

第一次打开 Genymotion 会要求输入之前注册的用户名和密码，然后会询问是否添加一台新的

虚拟机，选择“yes”。之后会出现支持的设备列表，选择需要安装的设备 and 系统版本，一直单击“next”按钮，直到开始下载模拟器，请保证整个过程不能断网。下载完毕之后就能够在设备列表中看到刚才所选择的设备信息，如图 11.23 所示。接下来单击“start”按钮启动安卓模拟器，使用安卓模拟器中自带浏览器即可测试页面，如图 11.24 所示。

注意：Genymotion 允许把 APK 文件直接拖入模拟器中进行安装，然后查看 APP 中内嵌的 HTML 页面。

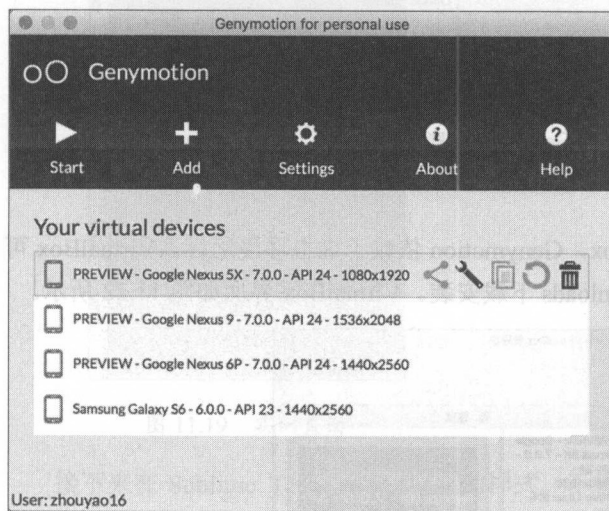


图 11.23 Genymotion 界面实例



图 11.24 Genymotion 模拟器浏览 Web 页面

11.4.2 iOS 模拟器调试

iOS 模拟器需要借助 Mac OS 系统下的 Xcode 工具。前往其官网下载 Xcode，地址为 <https://developer.apple.com/cn/xcode/>。Xcode 界面如图 11.25 所示。

启动 Xcode 之后，单击“Create a new Xcode project”按钮新建项目，选择“Single View Application”，然后单击“Next”，填上项目名称后再单击“Next”，接着选择存放的文件夹结束流程。在左上角的设备列表中选择需要的模拟器版本，这里选择 iPhone 7 Plus 设备，如图 11.26 所示。

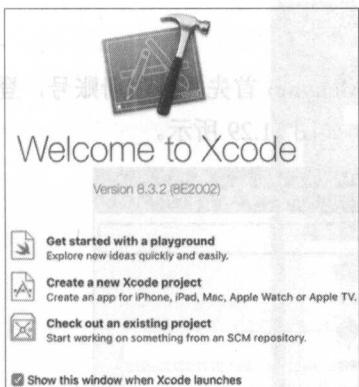


图 11.25 Xcode 界面



图 11.26 选择 iPhone 7 Plus 作为测试项目设备

Xcode 可选的设备列表，如图 11.27 所示。选择好设备之后，单击左上角的“Build”按钮启动项目。项目启动后会出现一个空白页面的“Simulator”，然后按下“shift+command+h”退回到模拟器桌面，接着打开模拟器桌面的 Safari 浏览器，输入需要测试的页面的地址，查看页面的功能和效果，如图 11.28 所示。

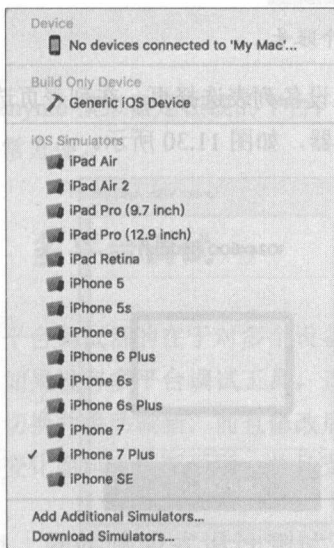


图 11.27 设备列表



图 11.28 iOS 模拟器

11.4.3 在线模拟器 Manymo

本节将介绍一款在线 Android 模拟器 Manymo。使用 Manymo 首先需要注册账号，登录官网 <https://www.manymo.com/>，完成注册，记住用户名和密码，如图 11.29 所示。

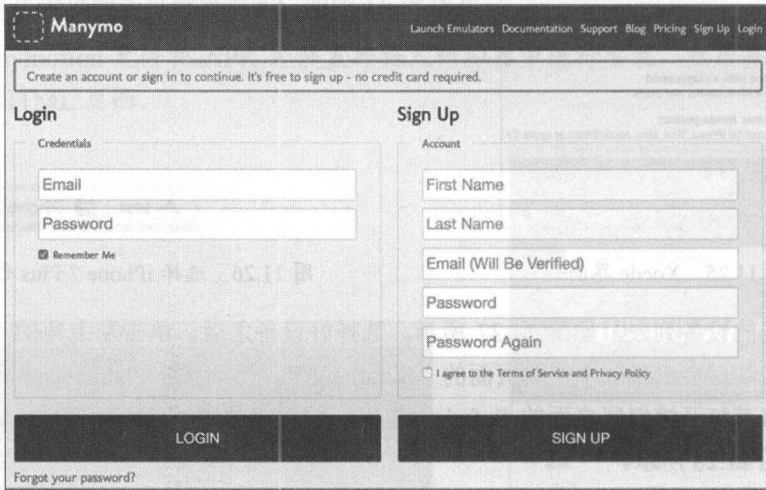
The image shows the Manymo website's login and sign-up interface. At the top, there's a navigation bar with links: Launch Emulators, Documentation, Support, Blog, Pricing, Sign Up, and Login. Below this, a message states: "Create an account or sign in to continue. It's free to sign up - no credit card required." The interface is split into two main sections: "Login" and "Sign Up". The "Login" section has fields for "Email" and "Password", a "Remember Me" checkbox, and a "LOGIN" button. Below it is a link for "Forgot your password?". The "Sign Up" section has fields for "First Name", "Last Name", "Email (Will Be Verified)", "Password", and "Password Again", along with an "I agree to the Terms of Service and Privacy Policy" checkbox and a "SIGN UP" button.

图 11.29 在 Manymo 上注册一个账号

登录之后，单击导航栏中的“Launch Emulators”，进入设备列表选择页。在列表页选择需要的分辨率和系统版本，然后单击“LAUNCH”按钮启动模拟器，如图 11.30 所示。

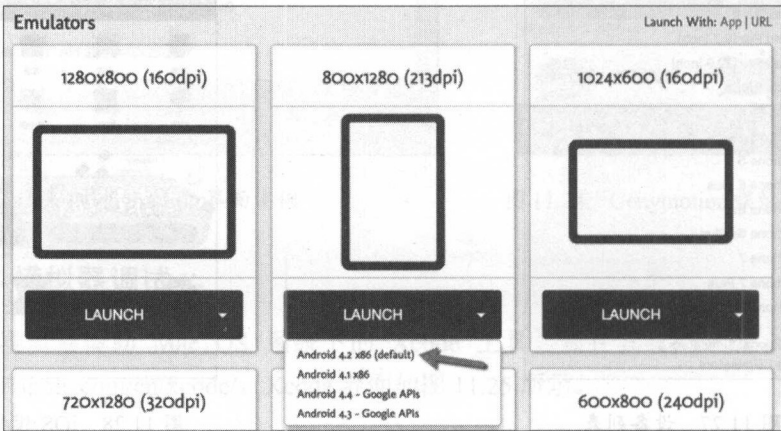


图 11.30 选择需要的分辨率和系统版本

启动了模拟器之后，通过模拟器内置浏览器查看页面的功能和样式效果，如图 11.31 所示。



图 11.31 Manymo 模拟器

Manymo 模拟器是在线的平台，无论你是 Windows 系统还是 Mac OS 系统均可使用，并且免费，非常方便。

11.5 多平台调试

多平台调试目的在于对多个设备测试和调试，主要解决兼容性问题和提高测试、调试的工作效率。如果没有多平台调试工具，查看一个页面的显示和使用是非常烦琐的，必须要在多个设备中来回切换并手动刷新，而且修改后也需要手动刷新查看最新效果。多平台调试往往能监控本地文件的变化而自动刷新页面，并且支持多设备同步刷新。

11.5.1 网站响应式设计测试工具 Ghostlab

Ghostlab 是一款 Mac 应用程序，用于在多设备上进行站点和 Web App 的同步化测试。虽然 Ghostlab 的初衷是用于测试，但是用于调试页面也是很好的工具。Ghostlab 官网地址为 <https://www.vanamco.com/>。Ghostlab 是一款收费软件，但是可以免费试用 7 天。Ghostlab 使用简

单, 支持监控本地目录、多设备同步刷新。

下载安装 Ghostlab 之后, 打开应用程序, 首次进入会出现一个提示弹窗, 选择 “Start Demo” 打开应用程序主界面, 如图 11.32 所示。

主界面顶部的黑色区域有功能的按钮分别是增加和删除需要测试的页面、搜索、界面切换以及帮助。黑色区域下方是测试页面的列表。单击黑色区域左上角的 “+” 按钮, 添加需要测试的页面地址, 如图 11.33 所示。

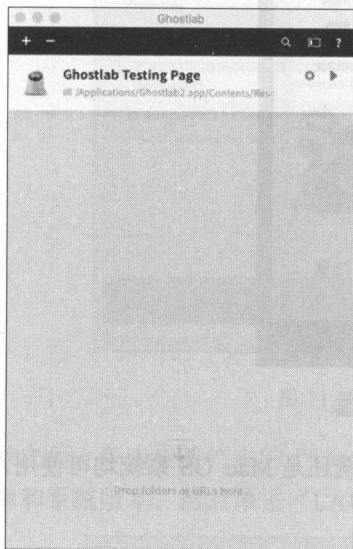


图 11.32 Ghostlab 主界面



图 11.33 添加需要测试的页面

页面添加成功之后, 单击页面列表右侧的 “三角” 按钮启动调试, 然后单击 “open with browser” 来添加需要测试或调试的设备, 也可以通过生成的二维码使用移动设备, 如图 11.34 所示。

添加完浏览器后开始调试, 操作任意浏览器任意页面 (包括滚动、填写表单等) 都会同步到其他浏览器。如果是本地的项目, 本地修改保存后, 所有浏览器都会同步刷新。除了同步以外, 单击浏览器列表右侧的 “inspect this client” 还能够查看页面源码, 如图 11.35 所示。



图 11.34 添加需要测试或调试的浏览器

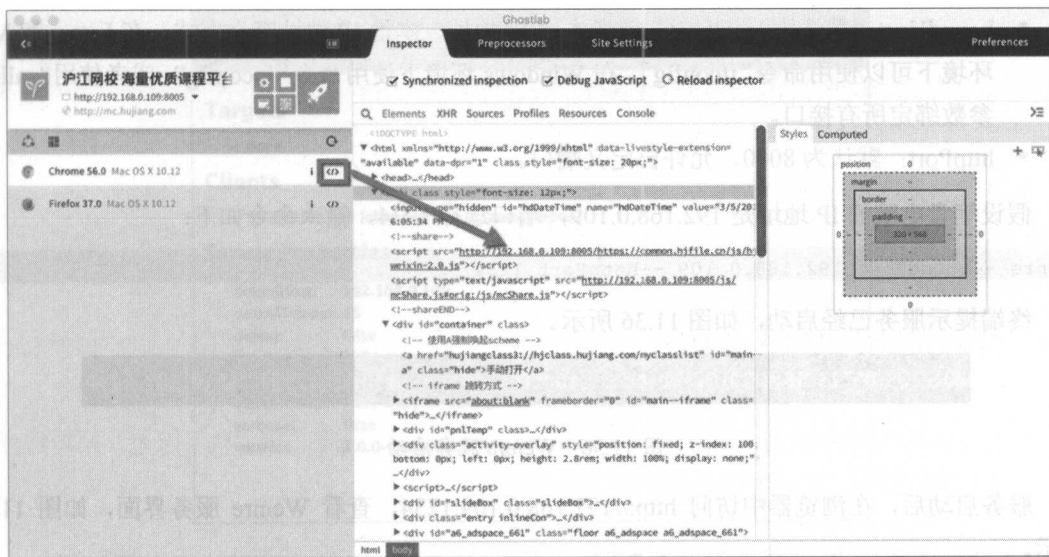


图 11.35 调试页面源码

图 11.35 的调试界面和 Chrome 的开发者工具非常类似，调试方法基本相同。在 Inspector 中的任何修改也会同步到所有浏览器和设备中。以上是 Ghostlab 的最主要的用法，还可以配合其他工具做一些更酷的事情，大家可以多尝试。

11.5.2 移动端 Web 开发调试工具 Weinre

在移动端页面的日常开发中常常会碰到这样一个问题，页面在 Chrome 的 Device 模式下显示和运行正常，但是如果页面在移动端浏览器或者内嵌到 APP 里就会出现样式问题。可是，移动端上没有类似 Chrome 的开发者调试工具，只能通过尝试修改，重复发布来检查问题，或者是写上一大堆“alert”弹窗来调试。这样的做法不但烦琐，而且收效甚微。Weinre 这个工具很好地解决了这个问题。

Weinre 是一款依赖于 Node.js 的远程调试工具，要使用 Weinre 进行调试，需要先安装 Node.js 环境以及 NPM，请参考“第 2 章 移动 Web 开发环境搭建”相关内容。使用 NPM 安装 Weinre，命令如下：

```
npm -g install weinre
```

安装完 Weinre 之后，在终端中输入命令启动 Weinre，命令如下：

```
weinre --boundHost [IP 地址] --httpPort [端口]
```

- **boundHost**: 默认为 localhost，或者本地 IP 地址。本地 IP 地址查询方式，在 Linux 和 Mac 环境下可以使用命令“ifconfig”，在 Windows 环境下使用命令“ipconfig”。或者使用“-all-”参数绑定所有接口。
- **httpPort**: 默认为 8080，允许自定义端口。

假设笔者电脑的 IP 地址是 192.168.0.109，端口选择 1234，输入命令如下：

```
weinre --boundHost 192.168.0.109 --httpPort 1234
```

终端提示服务已经启动，如图 11.36 所示。

```
2017-03-05T13:48:07.163Z weinre: starting server at http://192.168.0.109:1234
```

图 11.36 Weinre 服务启动

服务启动后，在浏览器中访问 <http://192.168.0.109:1234>，查看 Weinre 服务界面，如图 11.37 所示。

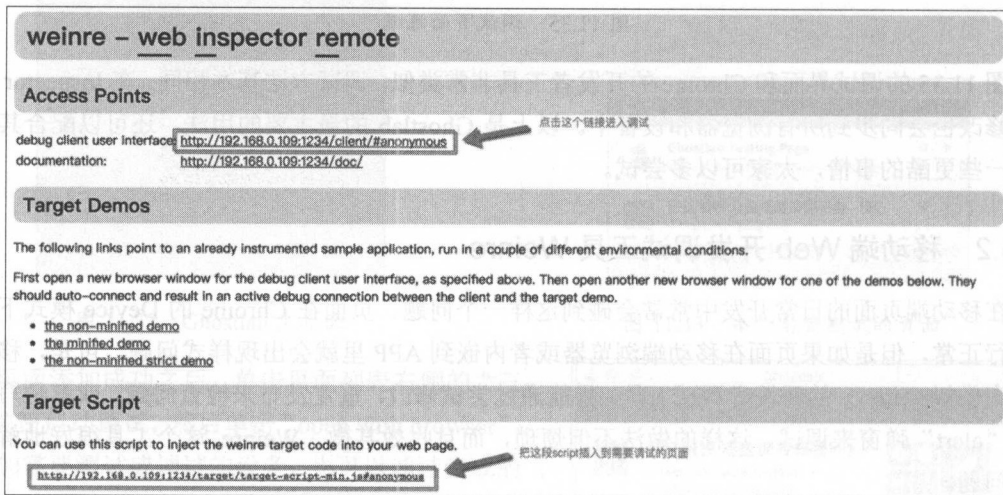


图 11.37 Weinre 服务界面

如图 11.37，Weinre 服务启动后，首先需要将“Target Script”标题下的 JavaScript 插入需要调试的页面，然后打开“Access Points”标题下“debug client user interface”选项对应的地址进入调试界面。调试界面如图 11.38 所示。

打开插入 Weinre 调试代码的目标页面，Weinre 调试界面同步出现一个新的 Target，单击新的 Target 进入调试，如图 11.39 所示。

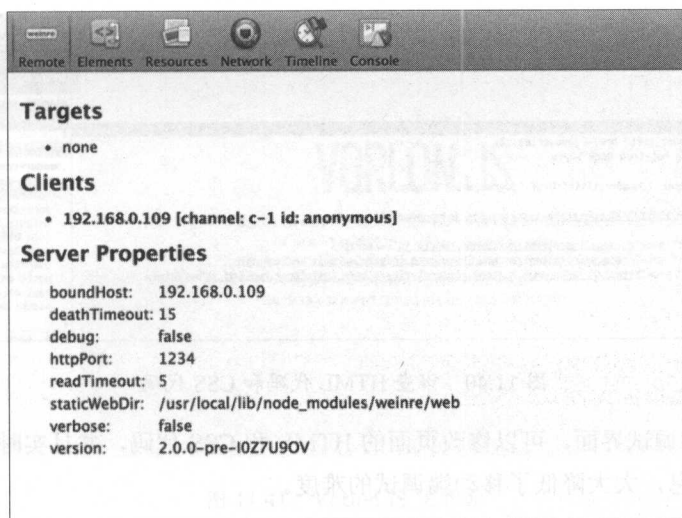


图 11.38 Weinre 调试界面

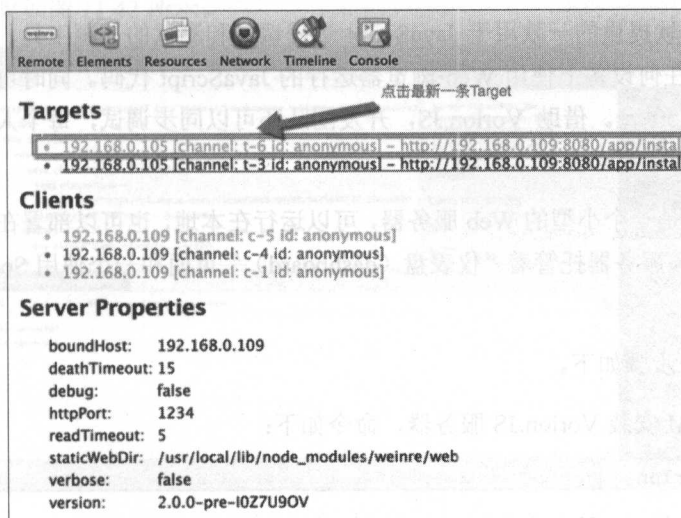


图 11.39 Weinre 调试界面中生成新的 Target

接下来的使用同 Chrome 开发者工具类似，比如单击“Elements”选项能够审查 HTML 代码和 CSS 代码，如图 11.40 所示。

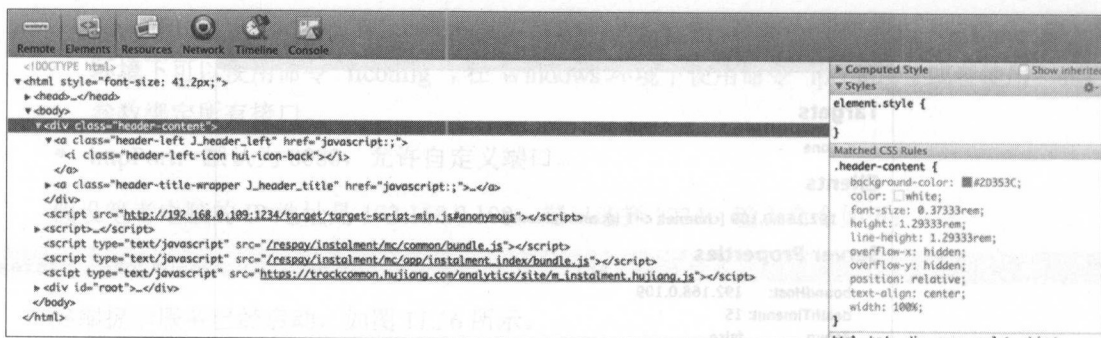


图 11.40 审查 HTML 代码和 CSS 代码

通过 Weinre 的调试界面，可以修改页面的 HTML 和 CSS 代码，并且实时预览，还能够查看控制台中的调试信息，大大降低了移动端调试的难度。

11.5.3 JavaScript 远程调试和测试工具 Vorlon.JS

Vorlon.JS 是微软提供的一款用于 JavaScript 远程调试和测试的开源工具，帮助开发者加载、检查、测试及调试任何设备上使用 Web 浏览器运行的 JavaScript 代码。同时可以连接各种设备，设备数量最多可达 50 台。借助 Vorlon.JS，开发团队还可以同步调试，每个人编写的代码都对所有人可见。

Vorlon.JS 本身是一个小型的 Web 服务器，可以运行在本地，也可以部署在一台服务器上供整个团队一起使用。该服务器托管着“仪表盘（dashboard）”页面和一个使用 Socket.IO 连接该页面及各种设备的服务。

安装 Vorlon.JS 步骤如下。

（1）使用 NPM 安装 Vorlon.JS 服务器，命令如下：

```
npm install -g vorlon
```

（2）安装完成之后，输入命令“vorlon”启动 Vorlon.JS 的服务。

Vorlon.JS 服务启动之后，在调试页面插入 JavaScript 代码。IP_address 默认为 localhost，允许使用本地 IP 地址。格式如下：

```
<script src="http://[IP_address]:1337/vorlon.js"></script>
```

（3）在浏览器中打开地址 [http://\[IP_address\]/dashboard/default](http://[IP_address]/dashboard/default)。页面如图 11.41 所示。

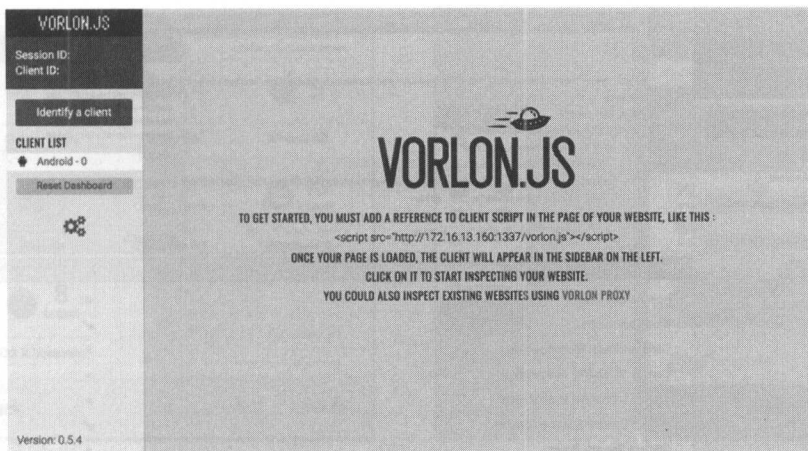


图 11.41 Vorlon.JS 主界面

(4) 实例中为安卓手机，在 Vorlon.JS 主界面的左侧可以看到“Android-0”的链接，单击该链接开始调试，效果如图 11.42 所示。

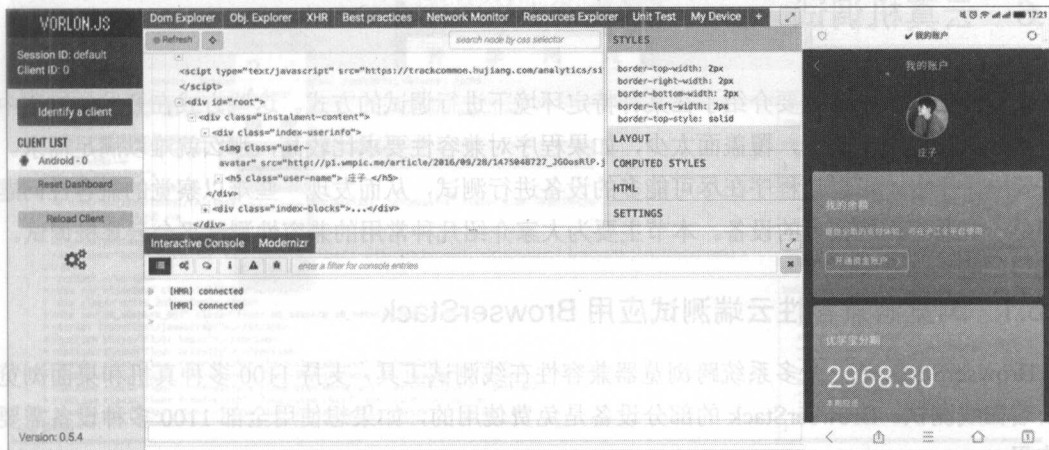


图 11.42 Vorlon.js 调试

调试方式和 PC 浏览器中的开发者工具相同，如图 11.43 所示。

可以通过“Dom Explorer”标签页调试页面的 DOM 结构和 CSS 样式。可以在“XHR”标签页中调试 AJAX 接口，可以在“Interactive Console”标签页中查看控制台信息。还有一个令人激动的功能，Vorlon.JS 自带特性检测，通过“Modernizr”标签页可以查看页面特性检测结果，帮助开发者快速定位页面兼容性问题。

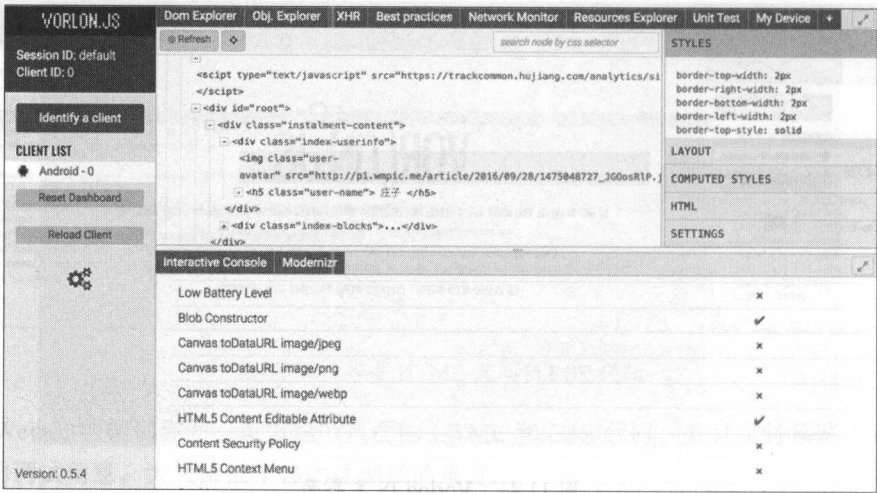


图 11.43 Vorlon.JS 特性检测

11.6 云真机调试

在前面的章节中，主要介绍了在某种特定环境下进行调试的方式。这些方式在开发的过程中非常有用，但是有一个缺点，覆盖面太少，如果程序对兼容性要求比较高，那么就难以满足需求。所以需要有一个平台，可以让程序在尽可能多的设备进行测试，从而发现一些难以察觉的兼容性问题，特别是一些兼容问题较多的设备。本节主要为大家介绍几种常用的兼容性测试平台云真机调试。

11.6.1 浏览器兼容性云端测试应用 BrowserStack

BrowserStack 是一个多系统跨浏览器兼容性在线测试工具，支持 1100 多种真机和桌面浏览器的云端在线测试。BrowserStack 的部分设备是免费使用的，如果想使用全部 1100 多种设备需要花钱购买。

BrowserStack 虽然是一个在线平台，但同时支持安装 Chrome 浏览器插件，插件效果如图 11.44 所示。单击面板中所需要测试的设备，选择其中一个尺寸，如图 11.45 所示。

在左侧的工具栏中还有很多非常实用的功能，如刷新页面、截图、对页面做一些设置。令人激动的是，BrowserStack 还提供了“DEVTOOLS”，可以直接在浏览器中对页面进行调试，不用借助于第三方工具。而且，BrowserStack 提供的“DEVTOOLS”与 Chrome 的使用方法相同，如图 11.46 所示。

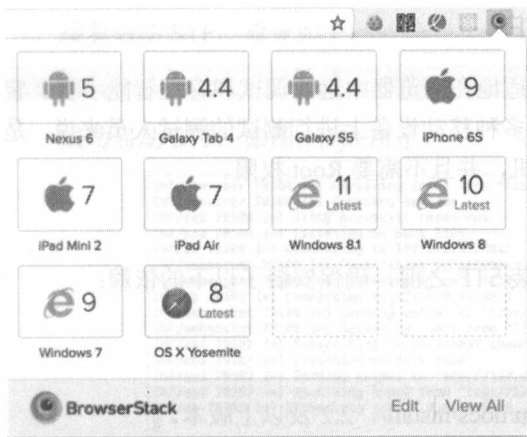


图 11.44 BrowserStack 的 Chrome 扩展程序面板



图 11.45 安卓真机测试

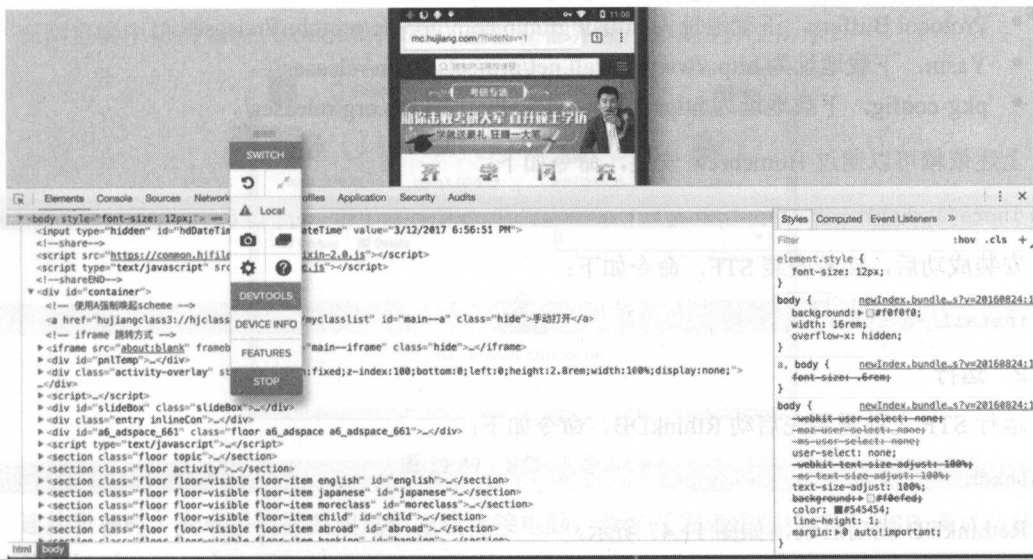


图 11.46 BrowserStack 提供的 devtools

除了可以使用移动真机来测试页面外，也可以使用不同操作系统的桌面浏览器来测试页面。如果平常使用 Mac OS 来开发程序，就可以通过这个平台来测试程序在 Internet Explorer 浏览器下的运行情况。如果平常使用 Windows 系统来开发程序，也可以通过这个平台测试程序在 Safari 浏览器中的运行情况。

11.6.2 Web 端移动设备管理控制工具 STF

STF (Smartphone Test Farm) 是一个可以很舒适地在浏览器中远程调试和管理智能手机、智能手表和其他小工具的 Web 应用程序。对于需要在多种移动设备上进行测试的测试人员来说，是一个非常不错的工具。目前支持安卓系统的智能手机，并且不需要 Root 权限。

1. 安装

下面介绍在 Mac OS 上安装 STF 的方法。在安装 STF 之前，确保安装了以下的依赖：

- Node.js, 6.9 及以上的版本。
- ADB, 安卓设备调试桥。
- RethinkDB, 下载地址为 <https://rethinkdb.com/docs/install/>, 2.2 及以上版本。
- GraphicsMagick, 下载地址为 <http://www.graphicsmagick.org/download.html>。
- ZeroMQ, 下载地址为 <http://zeromq.org/intro:get-the-software>。
- Protocol Buffers, 下载地址为 <https://github.com/google/protobuf/releases>。
- Yasm, 下载地址为 <http://www.tortall.net/projects/yasm/releases/>。
- pkg-config, 下载地址为 <https://pkg-config.freedesktop.org/releases/>。

上述依赖可以通过 Homebrew 安装，命令如下：

```
brew install rethinkdb graphicsmagick zeromq protobuf yasm pkg-config android-platform-tools
```

安装成功后，接着安装 STF，命令如下：

```
npm install -g stf
```

2. 运行

运行 STF 之前需要先启动 RethinkDB，命令如下：

```
Rethinkdb
```

RethinkDB 成功启动，如图 11.47 所示。

```
[~]- rethinkdb
Running rethinkdb 2.3.5 (CLANG 7.3.0 (clang-703.0.31))...
Running on Darwin 15.3.0 x86_64
Loading data from directory /Users/zoel/rethinkdb_data
warn: Cache size does not leave much memory for server and query overhead (available memory: 840 MB).
warn: Cache size is very low and may impact performance.
Listening for intracluster connections on port 29015
Listening for client driver connections on port 28015
Listening for administrative HTTP connections on port 8080
Listening on cluster addresses: 127.0.0.1, ::1
Listening on driver addresses: 127.0.0.1, ::1
Listening on http addresses: 127.0.0.1, ::1
To fully expose RethinkDB on the network, bind to all addresses by running rethinkdb with the '--bind all'
command line option.
Server ready, "admin_carl_hujiang_com_42j" a0df7ce1-d7b9-4222-9994-3901fd1b4588
```

图 11.47 成功启动 RethinkDB

接着启动 STF，命令如下：

```
stf local
```

成功启动 STF，如图 11.48 所示。

```
INF/provider 79104 [*] Receiving input from "tcp://127.0.0.1:7114"
INF/provider 79104 [*] Tracking devices
INF/app 79106 [*] Using pre-built resources
INF/app 79106 [*] Listening on port 7105
INF/db 79106 [*] Connecting to 127.0.0.1:28015
INF/websocket 79108 [*] Subscribing to permanent channel "ALL"
INF/websocket 79108 [*] Listening on port 7110
INF/db 79108 [*] Connecting to 127.0.0.1:28015
INF/websocket 79108 [*] Sending output to "tcp://127.0.0.1:7113"
INF/websocket 79108 [*] Receiving input from "tcp://127.0.0.1:7111"
INF/api 79107 [*] Subscribing to permanent channel "ALL"
INF/api 79107 [*] Listening on port 7106
INF/api 79107 [*] Sending output to "tcp://127.0.0.1:7113"
INF/api 79107 [*] Receiving input from "tcp://127.0.0.1:7111"
INF/db 79107 [*] Connecting to 127.0.0.1:28015
```

图 11.48 成功启动 STF

在浏览器中访问 <http://localhost:7100/>，查看设备界面，如图 11.49 所示。

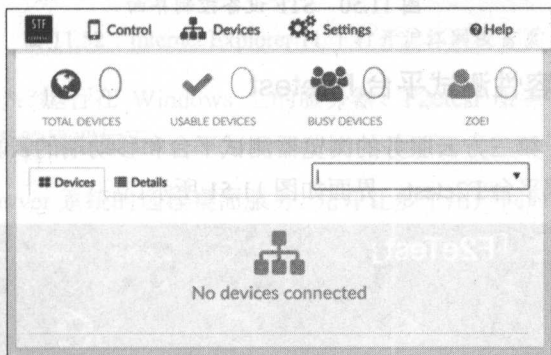


图 11.49 STF 设备界面

连接设备时，安卓手机需要通过数据线连接电脑，并在开发者选项中打开 USB 调试功能。初次连接手机时，STF 会在手机端安装 STF 工具，用户选择允许安装。安装成功后，在设备界面出现可用设备。单击某可用设备，进入设备的控制界面，如图 11.50 所示。

在控制界面，用户可以在左边的手机画面上通过鼠标和键盘远程控制手机，手机的画面会实时地传到电脑上，体验是否流畅。右边还有很多辅助的功能，如在手机上执行 Shell 脚本、截图、自动化功能、控制手机硬件按钮、管理文件等。更多详细的说明，请参考 STF 的官方介绍，地址为 <https://github.com/openstf/stf>。



图 11.50 STF 设备控制界面

11.6.3 多浏览器兼容性测试平台 F2etest

前面两节介绍了基于第三方云服务的浏览器测试平台和移动端的云真机测试平台，本节介绍基于开源的云浏览器测试平台 F2etest。界面如图 11.51 所示。



图 11.51 F2etest 主界面

在主界面中，列出了已经部署好的浏览器。单击需要进入的浏览器图标，打开云端浏览器，如图 11.52 所示。

可以很方便地在浏览器中远程访问云测试服务器上的浏览器测试页面，并进行兼容性测试。

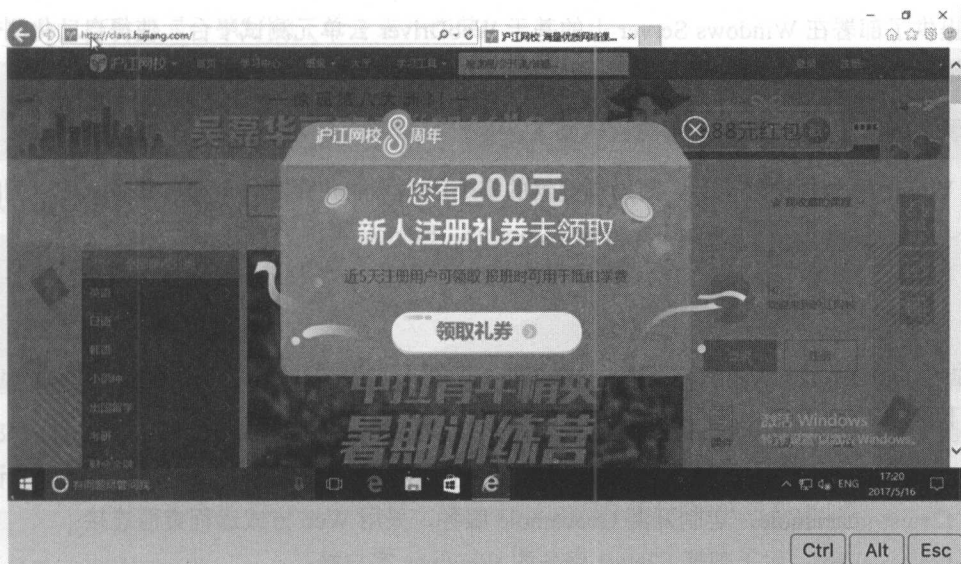


图 11.52 Internet Explorer 11 下打开沪江网校首页

该平台分为三个部分：运行在 Windows 上的服务器、F2etest 服务和 Guacamole（开源 Web 版远程桌面）。F2etest 系统原理如下。

- 利用 Windows Server 系统的远程桌面服务，允许让多个用户同时连接一台服务器进行操作，并且允许用户远程执行应用，打开对应的浏览器。
- 在 Windows Server 服务器上，部署一个 ASP 程序，通过 VB 脚本注册用户，在 F2etest 服务中，通过 API 调用部署在每一台 Windows Server 上的 ASP 程序，从而达到用户的集中管理。
- 通过 Guacamole 程序，采用 Web 的方式，连接 Windows Server 上的远程桌面服务，访问部署在 Windows Server 上的浏览器。
- 在 F2etest 服务中，提供代理服务，统一管理浏览器的代理，用户可以通过 F2etest 服务提供的代理接口，手动设置代理指向，可以将代理指向本地，利用本地配置的 Hosts 文件。

提示：Guacamole 是一个无客户端远程桌面网关。支持的标准协议有 VNC、RDP 和 SSH。不需要安装插件或客户端软件，可以直接在 Web 浏览器中使用，官网地址为 <https://guacamole.incubator.apache.org/>。

基于以上的原理，可以允许多个用户同时访问一台机器，高效地利用服务器资源。并且统一管理代理，使远端 Windows Server 采用本地设置的 Hosts 文件，提升了开发测试的便利。并且 F2etest

服务还提供了部署在 Windows Server 上的基于 WebDriver 云单元测试平台, 使得自动化测试也能很方便地运行在云测试服务器上。

部署 F2etest 服务, 需要准备的机器如下。

- Windows Server 若干台, 因为浏览器的多个版本不能共存在一个服务器中。可以同时在一台服务器上部署 Internet Explorer、Chrome、Firefox 等浏览器。
- Linux 服务器一台(或多台)用于部署 F2etest 服务和 Guacamole 服务, 在本节中将两个服务部署在同一台服务器上。

下面开始安装 F2etest, 从 GitHub 下载源码, 地址为 <https://github.com/alibaba/f2etest.git>, 项目包含以下目录。

- f2etest-web: F2etest 服务的 Web 站点, 用于提供 F2etest 服务;
- f2etest-guacamole: 定制开源 Guacamole 服务, 采用 Web 方式远程桌面连接;
- f2etest-client: 用于部署 F2etest 服务的 Windows 客户端;
- hostsShare-client: 用于修改远程 Windows 服务上的 Hosts。

部署 Windows 服务器, 这里介绍单台机器的部署, 其他机器可以参考此操作, 部署步骤如下。

- 安装远程桌面服务, 并对这个远程桌面服务进行激活。
- 部署 “/f2etest-client/f2etest-browsers/www/” 目录下的 setuser.asp 到 IIS 的 ASP 网站根目录下。并且设置 IIS 的执行权限为管理员权限, 以便有权限创建用户。
- 配置 Windows Server 为每天凌晨重启服务器, 避免出现服务器长时间运行后出现不稳定。
- 权限配置, 通过 F2etest 服务创建的用户默认为普通用户, 修改权限使得用户获得连接远程桌面服务权限。
- 配置 “f2etest-client/f2etest-browsers/curl/” 的 curl 命令。
- 安装浏览器。
- 配置运行浏览器的批处理文件, 参考 “f2etest-client/f2etest-browsers/app” 目录下的文件。

部署 Guacamole, 安装需要的依赖, 命令如下:

```
yum install cairo-devel libpng-devel uuid-devel freerdp* libvncserver-devel openssl-devel
```

创建软链, 命令如下:

```
ln -s /usr/local/lib/freerdp/guacsnd.so /usr/lib64/freerdp/  
ln -s /usr/local/lib/freerdp/guacdr.so /usr/lib64/freerdp/
```

进入 “f2etest/f2etest-guacamole/” 目录, 复制 guacamole-server-0.9.3.tar.gz 到安装目录, 解压

该文件，并且在此目录下执行如下命令进行安装：

```
./configure --with-init-dir=/etc/init.d
make
make install
```

执行注册服务，命令如下：

```
ldconfig
chkconfig --add guacd
chkconfig guacd on
chkconfig --list guacd
service guacd start
```

部署 Guacamole 客户端，将“f2etest/f2etest-guacamole/”下的 guacamole-0.9.3.war 部署到 Tomcat 中，新建配置文件如下：

```
mkdir /etc/guacamole
mkdir /root/.guacamole
vi /etc/guacamole/guacamole.properties
```

在配置文件中编辑如下内容，配置如下：

```
# Hostname and port of guacamole proxy
guacd-hostname: localhost
guacd-port: 4822
enable-websocket: true
enable-clipboard-integration: true
auth-provider: net.sourceforge.guacamole.net.auth.noauth.NoAuthenticationProvider
noauth-config: /etc/guacamole/noauth-config.xml
```

建立链接文件，命令如下：

```
ln -s /etc/guacamole/guacamole.properties /root/.guacamole
```

创建“/etc/guacamole/noauth-config.xml”文件，该文件配置远程访问服务器信息，内容如下：

```
<configs>
  <config name="f2etest-ie6" protocol="rdp">
    <param name="hostname" value="10.0.0.1" />           // Windows 服务器的 IP
    <param name="port" value="3389" />                   // 远程桌面的端口
    <param name="enable-drive" value="true" />           // 是否启动共享磁盘
    <param name="drive-path" value="/home/guacdshare" /> // 共享磁盘的地址
  </config>
  .....
</configs>
```

重新启动 Tomcat 服务后, Guacamole 客户端部署完毕。

最后, 部署 F2etest 服务, 步骤如下。

(1) 安装 Java、Tomcat、Node.js、MySQL。

(2) 从 GitHub 上下载该项目的源码, 项目地址为 <https://github.com/alibaba/f2etest.git>。

(3) 安装 f2etest-guacamole, 具体见 <https://github.com/alibaba/f2etest/blob/master/f2etest-guacamole/Install.md>。

(4) 创建 MySQL 数据库, 数据库名为“f2etest”。初始化数据库的脚本见“f2etest-web/f2etest.sql”。

(5) 进入 f2etest-web 目录, 执行“npm install”命令安装项目依赖。

(6) 配置通过 Guacamole 连接远程桌面的服务器信息, 修改文件“f2etest-web/conf/server.json”, 文件中的 ID 必须与先前配置的“/etc/guacamole/noauth-config.xml”文件中的名字一致。

(7) 配置通过 F2etest 访问的应用(浏览器)信息, 修改文件“f2etest-web/conf/app.json”, 这里的 Server 必须是 server.json 中配置的 ID, Program 为运行 Windows 服务器配置的批处理文件。

(8) 启动 Node.js 服务。

至此, 该服务基本已经配置完成, F2etest 项目官网地址为 <http://f2etest.com/>。

11.7 React 调试

11.7.1 React Developer Tools

React Developer Tools 是 Facebook 提供给开发者的用于调试使用 React 渲染的系统工具。可以在 Chrome 浏览器中通过添加“React Developer Tools”扩展来使用, 另外也有支持 Firefox、Atom/Nuclide 的插件及独立的 Electron 应用程序。本节将介绍在 Chrome 浏览器中安装和使用“React Developer Tools”。

(1) 前往 Chrome 网上应用店下载“React Developer Tools”插件, 如图 11.53 所示。



图 11.53 React Developer Tools

(2) 安装成功后，可以在在扩展程序界面（地址栏输入“chrome://extensions/”）查看新添加的“React Developer Tools”工具，启用并勾选“允许访问文件网址”选项，如图 11.54 所示。

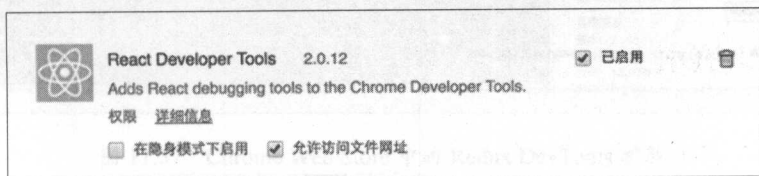


图 11.54 成功安装扩展

(3) 当在 Chrome 浏览器中打开一个使用 React 渲染的页面时，可以在浏览器的调试窗口看到 React 选项，单击 React 选项可以看到“React Developer Tool”界面，如图 11.55 所示。

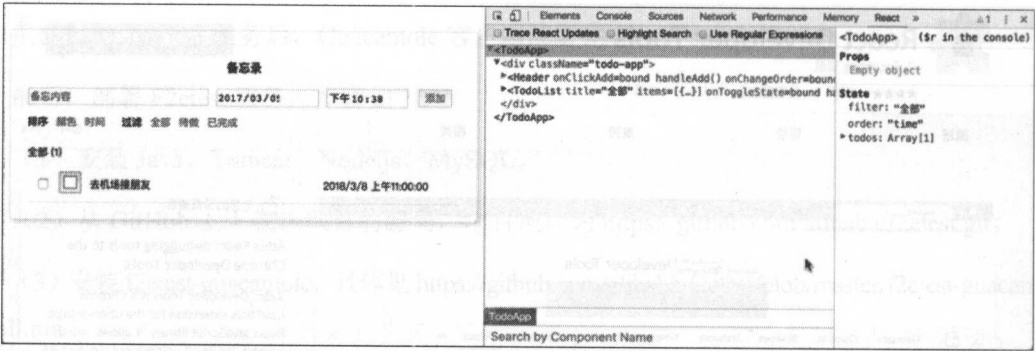


图 11.55 React Developer Tool 扩展界面

(4) 扩展界面的左半部分是组件树，区别于最终渲染的 DOM 树，通过组件树可以很直观地看出组件被渲染后对应的 UI。当把鼠标移到组件树中的某个组件节点时，可以在窗口左边的页面上看到对应的高亮 UI。在 Elements 选项中如果选中某个 DOM 元素，切换到 React 选项下时，也会自动选中 DOM 元素所对应的组件节点。相反，在组件节点上右击鼠标，在弹出的菜单中选择“Show in Elements Pane”也可以自动切换到 Elements 选项中并选中对应的 DOM 元素。选中某个组件时，也可以在控制台通过“\$r”查看组件信息，如图 11.56 所示。

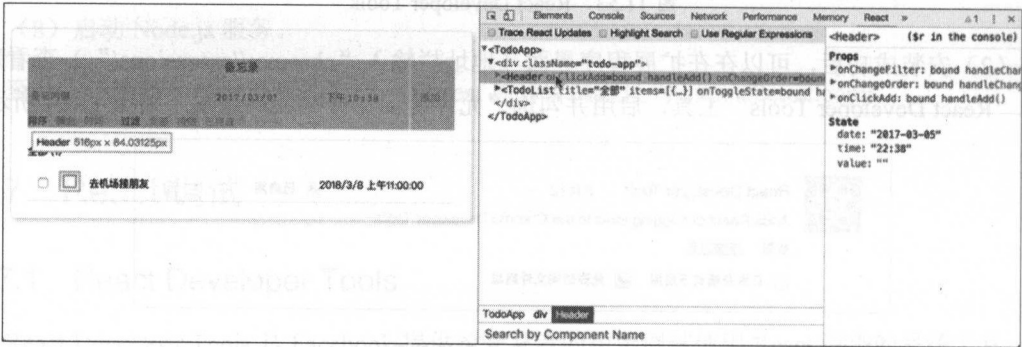


图 11.56 查看组件树对应的 UI

(5) 扩展界面的右半部分可以查看组件的 Props 和 State，当操作 UI 并引起状态变化时，也会及时地反映到显示的 State 数据上。还有一个更赞的功能，开发者随时可以修改 State 中的数据，并且会及时地将变化后的组件渲染到页面上。右击 Props 或 State 某个属性值，在弹出菜单中选择“Store as global variable”，可以在控制台中通过“\$tmp”查看对应的值。

“React Developer Tools”对于 React 系统的调试来说，是一个非常不错的工具，可以实时地观察组件的状态，了解组件与最终渲染出来的 DOM 节点的关系。

11.7.2 Redux DevTools

Redux DevTools 是一个 Redux 开发调试工具，可以对 Redux 应用的状态进行记录、回放、实时编辑等。Redux DevTools 有两种使用方式：一种是通过 NPM 模块安装，在代码中作为 Redux 的中间件引入，这种方式不限制使用的环境，在非浏览器的环境中也可以使用，另一种方式是通过在浏览器中安装相应的扩展使用，这种方式不需要项目中安装任何模块，可以保证代码的纯净，目前提供 Chrome、FireFox 及 Electron 的扩展。本节就以扩展的方式为例，介绍如何安装和使用 Redux DevTools。

(1) 前往 Chrome 网上应用店下载 Redux DevTools，如图 11.57 所示。



图 11.57 Chrome Web Store 中的 Redux DevTools 扩展

(2) 单击“添加至 CHROME”安装扩展。安装成功后，可以在扩展程序界面（地址栏输入“chrome://extensions/”）查看新添加的 Redux DevTools 扩展，启用并勾选“允许访问文件网址”，如图 11.58 所示。

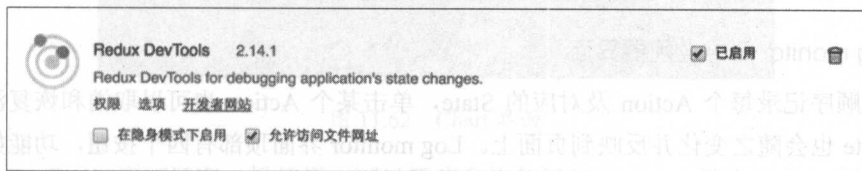


图 11.58 成功安装扩展

(3) 当在 Chrome 浏览器中打开一个使用 Redux 的应用时，可以在浏览器的调试窗口看到 Redux 选项，单击 Redux 选项可以看到 Redux DevTools 界面，如图 11.59 所示。

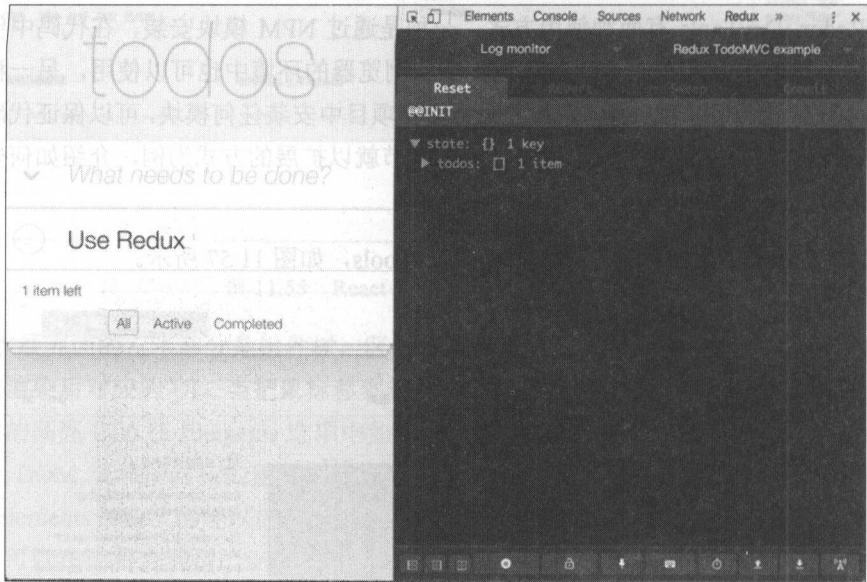


图 11.59 扩展界面

Redux DevTools 界面包含三个部分，如图 11.60 所示。

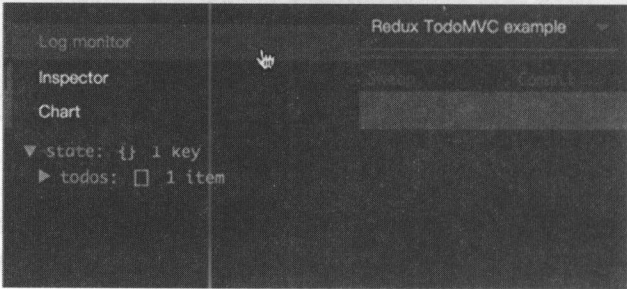


图 11.60 Redux DevTools 的三个部分

1. Log monitor 日志监视器界面

按时间顺序记录每个 Action 及对应的 State，单击某个 Action 也可以取消和恢复派发对应的 Action，State 也会随之变化并反映到页面上。Log monitor 界面顶部有四个按钮，功能如下。

Reset: 重置记录的 Action，恢复到初始状态。

- Commit: 提交当时的状态作为初始状态。
- Revert: 回滚 Action 到 Commit 后的初始状态。
- Sweep: 从日志中清除所有取消的 Action。

2. Inspector 检查器界面

检查每次操作的 Action 和 State, 还可以对两次操作进行差分比较、生成测试代码, 如图 11.61 所示。

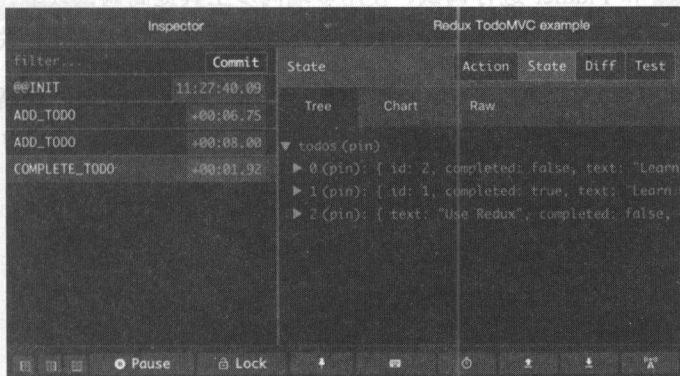


图 11.61 Inspector 界面

3. Chart 图表界面

以图表的形式表示 Store 中的 State, 如图 11.62 所示。

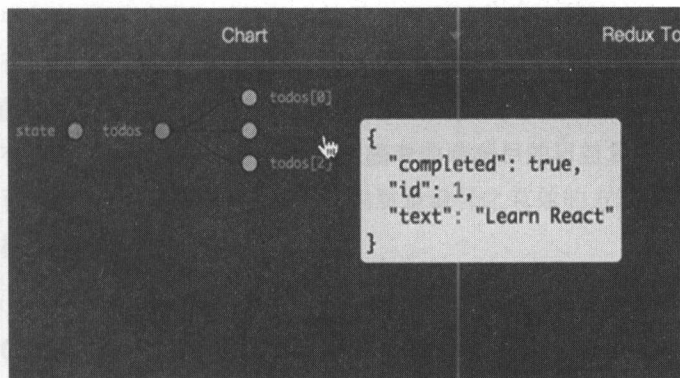


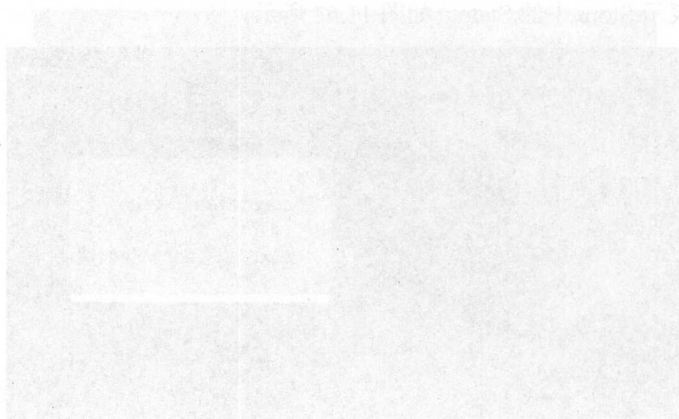
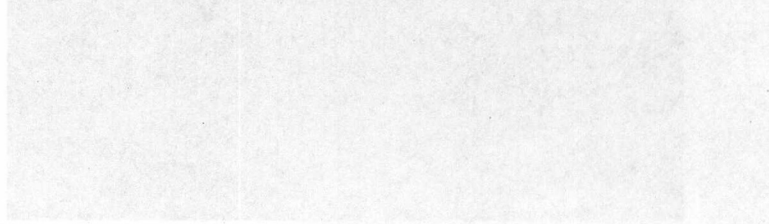
图 11.62 Chart 界面

Redux DevTools 底部还有一排按钮, 可以暂停和恢复记录 Action、锁定 State、派发自定义 Action 及时光旅行功能, 读者不妨亲自尝试一下。

11.8 本章小结

调试对于软件开发或者任何一项其他技术开发来说，都是必不可少的一个环节。调试就是在实际运行中去发现问题，解决问题，持续对产品进行优化，提高产品的质量。伴随着时代的进步，Web 前端业务越来越复杂，设备越来越多样化，技术越来越庞杂，调试的终端和工具也在不断发生变化。本章介绍了 Web 前端可能会使用到的一些调试工具，包括在浏览器中的开发者工具调试、使用代理工具 Charles 和 Fiddler 进行调试、使用多端同步工具提高调试效率、如何使用模拟器代替真实设备调试、利用不同平台的工具进行调试、在云真机上调试及 React 相关的一些调试方法。

优秀工具能帮助前端开发者更容易地发现隐患，更迅速地定位问题，提高开发效率，提高产品质量。期待读者能掌握本章介绍的工具和技术，并充分地运用到实际开发中。



12

第 12 章

前端单元测试实战

软件开发过程中的单元测试是指对软件中的最小可测试单元进行检查和验证，可以是一段用于测试一个模块是否能达到预期结果的代码。单元测试框架是可以为应用添加单元测试并运行单元测试的工具。在前端开发领域中，单元测试一直是一个被忽视的领域，早期大量的偏 UI 类代码使得前端开发并没有养成书写单元测试的习惯，但随着前端项目的日益复杂，包括 Node.js 的大量应用开发，为应用程序书写单元测试才是保证代码质量最行之有效的方法，本章将会介绍几种常用的前端单元测试框架。

12.1 JavaScript 单元测试框架 Jasmine 实战

Jasmine 是一个行为驱动开发模式的 JavaScript 单元测试框架，自身不依赖其他 JavaScript 类库，也不需要操作 DOM，并且同时支持运行在浏览器环境和 Node.js 环境。首先介绍 Jasmine 的安装使用。

(1) 创建一个空的应用程序，命令如下：

```
npm init
```

(2) 安装单元测试框架 Jasmine，命令如下：

```
npm install -g jasmine
```

(3) 初始化单元测试配置，命令如下：

```
jasmine init
```

(4) 生成单元测试样例，命令如下：

```
jasmine examples
```

(5) 运行单元测试，命令如下：

```
Jasmine
```

控制台输出运行结果，如图 12.1 所示。配置完成后的应用程序目录如图 12.2 所示。

- 目录/lib/jasmine_examples 包含需要书写单元测试的 JavaScript 业务模块。
- 目录/spec/jasmine_examples 包含为业务模块书写的 JavaScript 单元测试用例。
- 目录/spec/helpers 下包含单元测试执行前需要被预先执行的代码。

```
yiweilai@jasmine-test yiweilai$ jasmine
Started
*****
5 specs, 0 failures
Finished in 0.008 seconds
```

图 12.1 Jasmine 运行结果

```
└─ JASMINE-TEST
  └─ .vscode
  └─ lib
    └─ jasmine_examples
      ├── Player.js
      └── Song.js
  └─ node_modules
  └─ spec
    └─ helpers
      ├── jasmine_examples
      │   └── SpecHelper.js
      ├── jasmine_examples
      │   └── PlayerSpec.js
      └─ support
        ├── jasmine.json
        └── package.json
```

图 12.2 单元测试应用目录

文件“/spec/support/jasmine.json”配置了以上单元测试相关信息，配置如下：

```
{
  "spec_dir": "spec",          // 单元测试根目录
```

```

"spec_files": [                // 单元测试文件
  "**/*[sS]pec.js"
],
"helpers": [                   // 单元测试预执行 helper 文件
  "helpers/**/*.*.js"
]
}

```

下面结合 `Player.js` 模块的单元测试 `PlayerSpec.js` 来讲解 Jasmine 提供的单元测试功能。文件 `PlayerSpec.js` 代码如下:

```

01 describe("Player", function() {                // 一组单元测试
02     var Player = require('../../lib/jasmine_examples/Player'); // 引入业务模块
03     var Song = require('../../lib/jasmine_examples/Song');    // 引入业务模块
04     var player;                                               // 声明全局变量
05     var song;                                                 // 声明全局变量
06     beforeEach(function() {                                   // 每个单元测试前自动运行
07         player = new Player();                                // 实例化对象
08         song = new Song();                                    // 实例化对象
09     });
10     it("should be able to play a Song", function() {          // 一个单元测试实例
11         player.play(song);                                     // 调用业务模块函数
12         expect(player.currentlyPlayingSong).toEqual(song);    // 调用系统断言
13         expect(player).toBePlaying(song);                     // 调用自定义断言
14     });
15     describe("when song has been paused", function() {        // 一组嵌套的单元测试
16         beforeEach(function() {                                // 每个单元测试前自动运行
17             player.play(song);                                 // 播放歌曲
18             player.pause();                                    // 暂停播放
19         });
20         // 一个单元测试
21         it("should indicate that the song is currently paused", function() {
22             expect(player.isPlaying).toBeFalsy();              // 断言值为 false
23             expect(player).not.toBePlaying(song);              // 自定义断言值为 false
24         });
25         it("should be possible to resume", function() {
26             player.resume();                                    // 恢复播放
27             expect(player.isPlaying).toBeTruthy();             // 断言值为 true
28             expect(player.currentlyPlayingSong).toEqual(song); // 断言两值相等
29         });
30     });
31     // 一个单元测试

```



```

32     it("tells the current song if the user has made it a favorite", function() {
33         spyOn(song, 'persistFavoriteStatus');           // 监听某个函数
34         player.play(song);                               // 播放歌曲
35         player.makeFavorite();                           // 收藏歌曲
36         // 断言函数被调用
37         expect(song.persistFavoriteStatus).toHaveBeenCalledWith(true);
38     });
39     describe("#resume", function() {                     // 一组单元测试
40         // 一个单元测试
41         it("should throw an exception if song is already playing", function() {
42             player.play(song);                           // 播放歌曲
43             expect(function() {
44                 player.resume();                           // 恢复歌曲播放
45             }).toThrowError("song is already playing");   // 断言抛出异常
46         });
47     });
48 });

```

代码第 01 行定义了一组单元测试，代码第 02~03 行引入了需要进行单元测试的业务模块。

代码第 06~09 行定义了需要在每个单元测试运行前执行的代码。

代码第 10~14 行定义了一个单元测试，并使用了系统断言和自定义断言。

代码第 15~30 行定义了一组内嵌单元测试，并使用了断言来判断值为 `false` 或 `true`，以及两值是否相等。

代码第 31~38 行定义了一个单元测试，并使用了监听函数功能来判断函数是否被调用。

代码第 39~47 行定义了一组内嵌单元测试，并使用断言来判断是否有异常抛出。

通过以上代码分析，单元测试框架 Jasmine 的基本用法已经介绍完毕，关于如何自定义断言可以参考本书源码。如果想了解 Jasmine 的详细功能，可以参考其官网 <https://jasmine.github.io/2.5/introduction>。

12.2 使用 Mocha 和 Chai 在 Node.js 进行单元测试

Mocha 是一个功能丰富的 JavaScript 单元测试框架，对异步单元测试非常友好，并且同时支持运行在浏览器环境和 Node.js 环境。Chai 是一个 BDD/TDD 模式的断言库，也同时支持运行在浏览器环境和 Node.js 环境。因为 Mocha 并不提供断言功能，所以 Mocha 和 Chai 可以搭配使用。

(1) 创建一个空的应用程序，命令如下：

```
npm init
```

(2) 安装单元测试框架 Mocha，命令如下：

```
npm install -g mocha
```

(3) 安装断言库 Chai 以及 Chai 的插件 chai-spies，命令如下：

```
npm install chai chai-spies --save-dev
```

创建 JavaScript 业务模块以及对应的单元测试，具体内容见源码。创建完成后的应用目录如图 12.3 所示。

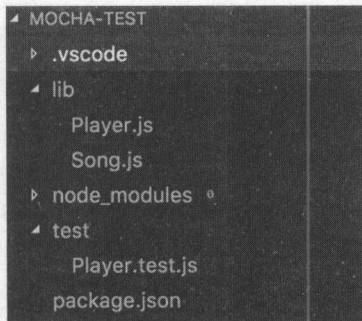


图 12.3 单元测试应用目录

目录 lib 包含的是需要书写单元测试的 JavaScript 业务模块，目录 test 包含的是为业务模块书写的 JavaScript 单元测试用例。运行单元测试。打开命令行窗口，输入如下命令：

```
Mocha
```

控制台输出运行结果，如图 12.4 所示。

```
yiweilai@mocha-test yiweilai$ mocha

Player
  ✓ should be able to play a Song
  ✓ tells the current song if the user has made it a favorite
  when song has been paused
    ✓ should indicate that the song is currently paused
    ✓ should be possible to resume
  #resume
    ✓ should throw an exception if song is already playing

5 passing (17ms)
```

图 12.4 Mocha 运行结果

下面结合 `Player.js` 模块的单元测试 `Player.test.js` 来讲解 Mocha 提供的单元测试功能。文件 `Player.test.js` 代码如下:

```

01 describe("Player", function() {                                // 一组单元测试
02     var Player = require('../lib/Player');                      // 引入业务模块
03     var Song = require('../lib/Song');                          // 引入业务模块
04     var chai = require('chai');                                // 引入断言库
05     var spies = require('chai-spies')                          // 引入断言库插件提供函数监视功能
06     var expect = chai.expect;                                  // 使用断言库 Chai 的 TDD 风格
07     var player;                                                // 声明全局变量
08     var song;                                                  // 声明全局变量
09     chai.use(spies);                                            // 注册断言库插件
10     beforeEach(function() {                                    // 每个单元测试前自动运行
11         player = new Player();                                  // 实例化对象
12         song = new Song();                                     // 实例化对象
13     });
14     it("should be able to play a Song", function() {           // 一个单元测试实例
15         player.play(song);                                     // 调用业务模块函数
16         expect(player.currentlyPlayingSong).to.be.equal(song); // 使用断言库
17     });
18     describe("when song has been paused", function() {        // 一组嵌套单元测试
19         beforeEach(function() {                                // 每个单元测试前自动运行
20             player.play(song);                                  // 播放歌曲
21             player.pause();                                     // 暂停播放
22         });
23         // 一个单元测试
24         it("should indicate that the song is currently paused", function() {
25             expect(player.isPlaying).to.be.false;              // 断言值为 false
26         });
27         it("should be possible to resume", function(done) {    // 一个单元测试
28             player.resume();                                    // 恢复播放
29             setTimeout(function() {                             // Mocha 自动支持异步测试
30                 expect(player.isPlaying).to.be.true;           // 断言值为 true
31                 done();                                         // 结束单元测试
32             });
33             expect(player.currentlyPlayingSong).to.be.equal(song); // 断言两值相等
34         });
35     });
36     // 定义一个单元测试
37     it("tells the current song if the user has made it a favorite", function() {
38         // 监听函数是否被调用

```

```

39     var persistFavoriteStatus = chai.spy.on(song, 'persistFavoriteStatus');
40     player.play(song); // 播放歌曲
41     player.makeFavorite(); // 收藏歌曲
42     expect(persistFavoriteStatus).to.have.been.called(); // 断言函数被调用
43   });
44   describe("#resume", function() { // 一组嵌套单元测试
45     // 定义一个单元测试
46     it("should throw an exception if song is already playing", function() {
47       player.play(song); // 播放歌曲
48       expect(function() {
49         player.resume(); // 恢复播放
50       }).to.throw("song is already playing"); // 断言抛出异常
51     });
52   });
53 });

```

代码第 01 行定义了一组单元测试，代码第 02~09 行引入了需要进行单元测试的模块以及断言库相关模块。

代码第 10~13 行在每个单元测试运行前实例化业务对象。

代码第 14~17 行定义了一个单元测试，并使用了断言库。

代码第 18~35 行定义了一组内嵌单元测试，并使用断言来判断值为 `false` 或 `true`，以及两值是否相等，同时使用了 Mocha 提供的异步单元测试功能。

代码第 37~43 行定义了一个单元测试，并使用了 Chai 断言库插件 `chai-spies` 的监听函数功能来判断函数是否被调用。

代码第 44~52 行定义了一组内嵌单元测试，并使用断言来判断是否有异常抛出。

通过以上代码分析，单元测试框架 Mocha 和断言库 Chai 的基本用法已经介绍完毕。单元测试框架 Mocha 的详细功能可以参考其官网 <https://mochajs.org/>，断言库 Chai 的详细功能可以参考其官网 <http://chaijs.com/>。

12.3 使用 Sinon 辅助单元测试

在书写单元测试的时候，碰到的最大的困难就是业务代码运行在测试环境，不方便模拟和控制资源，比如业务代码会发送网络请求、操作本地存储或者异步执行代码。Sinon 的功能就是在单

元测试运行时替换这些复杂操作，简化单元测试环境。

Sinon 只是一个用于单元测试的模拟库，需要配合其他单元测试框架使用。在本节的例子中，选择的单元测试框架是 Jasmine，关于 Jasmine 的使用方法请参考章节 12.1，此处不再赘述。

注意：使用 Sinon 前需要先通过 NPM 的方式安装到应用。

Sinon 通过 spy 方法来收集函数的调用信息。例如函数是否被调用，是哪个对象在调用函数，以及函数调用时传递的参数。单元测试代码如下：

```
it("spies", function() { // 定义一个单元测试
  var spy = sinon.spy(player, "play"); // 监听函数
  player.play(song); // 调用函数
  expect(player.isPlaying).toBeTruthy(); // 断言歌曲正在播放
  expect(spy.called).toBeTruthy(); // 断言函数被调用
  expect(spy.calledOn(player)).toBeTruthy(); // 断言函数被某个对象调用
  expect(spy.calledWith(song)).toBeTruthy(); // 断言函数调用时的参数
});
```

Sinon 通过 stub 方法来替换目标函数的调用，stub 方法拥有 spy 方法的全部功能。单元测试代码如下：

```
01 it("stub", function() { // 定义一个单元测试
02   var stub = sinon.stub(player, "play"); // 替换目标函数
03   player.play(song); // 调用函数
04   expect(player.isPlaying).toBeFalsy(); // 断言歌曲没有被播放
05   expect(stub.called).toBeTruthy(); // 断言函数被调用
06   expect(stub.calledOn(player)).toBeTruthy(); // 断言函数被某个对象调用
07   expect(stub.calledWith(song)).toBeTruthy(); // 断言函数调用时的参数
08   stub.restore(); // 不再替换目标函数
09   player.play(song); // 调用函数
10   expect(player.isPlaying).toBeTruthy(); // 断言歌曲正在播放
11 });
```

代码第 02~07 行使用 stub 方法替换了目标函数，当目标函数被调用时，目标函数不会执行。代码第 08~10 行解除了对目标函数的替换，当目标函数被调用时，目标函数会被执行。

使用 stub 方法替换目标函数之后，一定要在本单元测试结束时解除，否则会对其他单元测试产生影响。除了手动解除，还有一种自动解除的方式，单元测试代码如下：

```
it("auto restore", sinon.test(function() { // 定义一个自动解除 Sinon 模拟的单元测试
  var stub = this.stub(player, "play"); // 替换目标函数
  player.play(song); // 调用函数
```

```

expect(player.isPlaying).toBeFalsy();           // 断言歌曲没有播放
expect(stub.called).toBeTruthy();               // 断言函数被调用
});

```

如果需要自动解除功能，需要将测试体作为 `sinon.test` 函数的参数，并且使用 `this.stub` 代替 `sinon.stub`。

Sinon 通过 `mock` 方法来替换整个对象，并且包含内置的断言，方便定义期望的结果和行为。单元测试代码如下：

```

it('mock', function() {                          // 定义一个单元测试
    var mock = sinon.mock(player);               // 模拟对象
    mock.expects('play').once();                 // 断言 play 函数只被调用一次
    player.play(song);                           // 调用 play 函数
    expect(player.isPlaying).toBeFalsy();         // 断言歌曲没有播放
    mock.verify();                               // 验证 Mock 内置断言
});

```

Simon 通过 `useFakeTimers` 方法来实现异步单元测试。比如，`Player` 对象的 `pause` 方法里使用了 `setTimeout` 来延迟执行，代码如下：

```

Player.prototype.pause = function() {            // 定义 pause 方法
    var self = this;                             // 保存 this 对象
    setTimeout(function() {                      // 延迟 100ms 执行
        self.isPlaying = false;                // 设置为 false
    }, 100);
};

```

单元测试代码如下：

```

it('fake timer', function () {                  // 定义一个单元测试
    clock = sinon.useFakeTimers();              // 模拟异步执行
    player.play(song);                          // 播放歌曲
    player.pause();                             // 暂停歌曲
    expect(player.isPlaying).toBeTruthy();       // 断言歌曲正在播放
    clock.tick(99);                             // 模拟等待 99ms
    expect(player.isPlaying).toBeTruthy();       // 断言歌曲正在播放
    clock.tick(1);                             // 模拟等待 1ms
    expect(player.isPlaying).toBeFalsy();       // 断言歌曲停止播放
    clock.restore();                            // 停止模拟异步行为
});

```

至此，Sinon 的基本功能已经介绍完毕，如果想了解 Sinon 的详细用法，请参考其官网

<http://sinonjs.org/>。

12.4 使用 Karma 自动化单元测试

Karma 是一个基于 Node.js 的 JavaScript 测试执行过程管理工具。用于测试所有主流的 Web 浏览器，也可集成到持续集成（Continuous integration，简称 CI）工具里面，或者和其他代码编辑器一起使用。

Karma 有以下特点：

- 在真实浏览器上测试，Karma 的设计可以运行在不同的浏览器上，不会出现跨浏览器的问题。
- 远程控制，在一些手动触发的测试用例中，可以通过命令行的方式对 Karma 进行控制，即可以直接将命令集成至 IDE 工具。
- 自动执行，Karma 提供了文件监控功能，当监控的文件发生变化时，自动执行测试流程，并打印结果。
- 可扩展，Karma 提供了插件机制，允许开发者自定义实现一些功能。

与绝大多数前端框架、工具一样，Karma 也基于 Node.js 开发，所以在安装 Karma 之前需要先安装 Node.js。

Karma 可以通过 NPM 进行安装，命令如下：

```
npm install -g karma-cli
```

在项目中安装 Karma 包，命令如下：

```
npm install --save-dev karma
```

安装好 Karma 后，会自动安装断言库 Jasmine。

由于断言库 Jasmine 依赖 jasmine-core 库，所以还必须安装 jasmine-core，命令如下：

```
npm install --save-dev jasmine-core
```

在工程目录中运行命令“karma init”来进行测试环境的初始化，然后按照指示完成每一步。初始化操作如图 12.5 所示。

初始化完成后，会在项目根目录中自动生成配置文件 karma.conf.js，最终文档结构如图 12.6 所示。

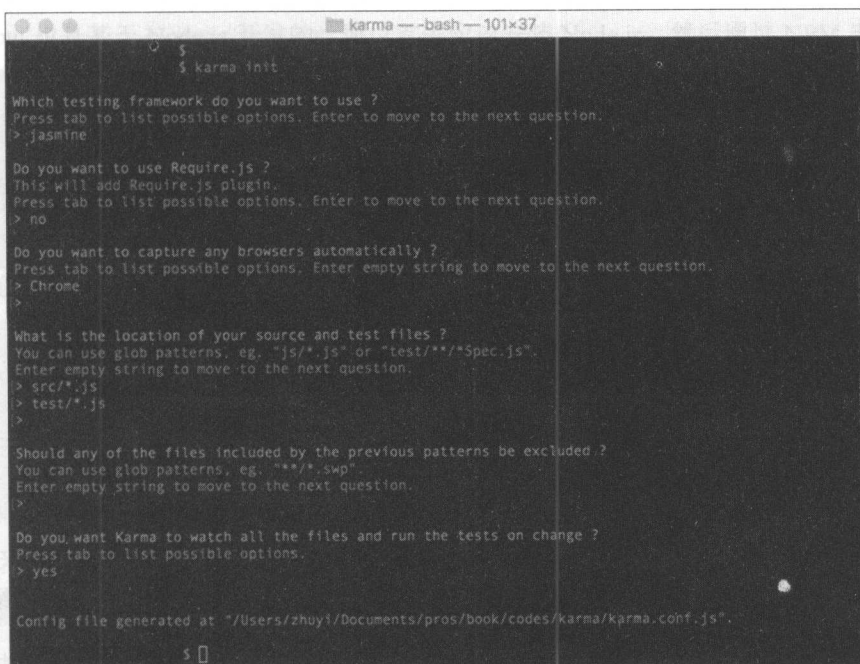


图 12.5 Karma 初始化测试环境截图

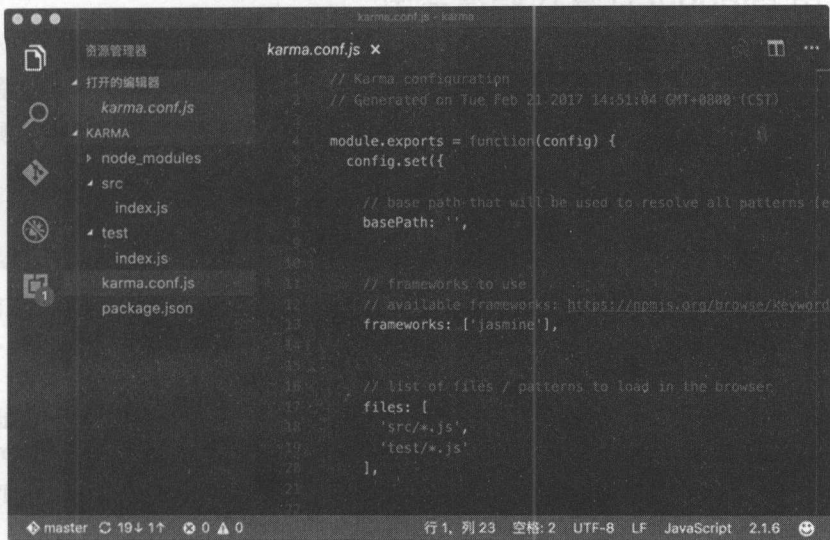


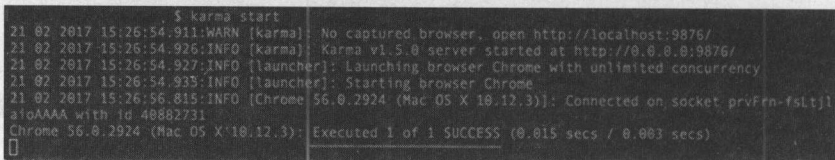
图 12.6 项目文档结构截图

至此，基本的 Karma 测试环境搭建完毕。接着，在 test/index.js 中编写一段简单的测试脚本，

代码如下：

```
describe('index.js: ', function() {
  it('test function isPhone(s).', function() {
    expect(isPhone('15212312412')).toBe(true);
    expect(isPhone('122333')).toBe(false);
  });
});
```

然后在控制台运行命令“`karma start`”，输出结果如图 12.7 所示。



```
$ karma start
21 02 2017 15:26:54.911:WARN [karma]: No captured browser, open http://localhost:9876/
21 02 2017 15:26:54.926:INFO [karma]: Karma v1.5.0 server started at http://0.0.0.0:9876/
21 02 2017 15:26:54.927:INFO [launcher]: Launching browser Chrome with unlimited concurrency
21 02 2017 15:26:54.933:INFO [launcher]: Starting browser Chrome
21 02 2017 15:26:56.815:INFO [Chrome 56.0.2924 (Mac OS X 10.12.3)]: Connected on socket prVFm-fsItj1aIoAAAA with id 40882731
Chrome 56.0.2924 (Mac OS X 10.12.3): Executed 1 of 1 SUCCESS (0.015 secs / 0.003 secs)
```

图 12.7 运行测试程序后的输出结果

通过控制台输出结果，可以看出测试程序已经执行成功，且测试通过。本文只介绍了 Karma 简略的使用，其他详细说明可以参考其官网 <https://karma-runner.github.io>。

12.5 使用 Istanbul 计算代码覆盖率

日常做代码测试时，开发者除了关注测试是否通过之外，还需关心测试是否全面，是否所有的代码都被覆盖，该指标即“代码覆盖率”（Code Coverage）。代码覆盖率有四个测量纬度，如下所示。

- 行覆盖率（Line Coverage）：是否每一行代码都被执行。
- 函数覆盖率（Function Coverage）：是否每一个函数都被调用。
- 分支覆盖率（Branch Coverage）：是否每一个 if/else 分支都执行到。
- 语句覆盖率（Statement Coverage）：是否每一句代码都被执行到。

本节要介绍的 Istanbul，就是计算 JavaScript 程序代码覆盖率的工具，GitHub 地址为 <https://github.com/gotwarlost/istanbul>。

提示：看到 Istanbul 这个单词，对西方城市有所了解的读者马上就会意识到，这不是土耳其的伊斯坦布尔吗？没错，这个单词正是伊斯坦布尔。因为土耳其的地毯世界闻名，而地毯就是作覆盖之用的，所以该项目就取了 Istanbul 这个名字。

Istanbul 也是基于 Node.js 开发的，所以首先应该安装 Node.js，然后通过 NPM 安装 Istanbul，命令如下：

```
npm install -g Istanbul
```

安装成功后，可以通过控制台命令 “istanbul help” 来查看 Istanbul 所支持的所有命令。接下来通过一个简短的实例来说明 Istanbul 的使用。

首先创建一个脚本文件 index.js，代码如下：

```
function add(a, b) {
  a = a || 0;
  b = b || 0;
  return a + b;
}
add(1, 2);
```

接下来打开控制台，进入 index.js 所在目录，输入以下命令：

```
istanbul cover index.js
```

得到执行结果如下：

```
=====
Writing coverage object [F:\pros\book\codes\istanbul\coverage\coverage.json]
Writing coverage reports at [F:\pros\book\codes\istanbul\coverage]
=====
===== Coverage summary =====
Statements   : 100% ( 5/5 )
Branches     : 50% ( 2/4 )
Functions    : 100% ( 1/1 )
Lines        : 100% ( 5/5 )
=====
```

输出结果显示，index.js 有 5 条语句，执行了 5 条；有 4 个分支，执行了 2 个；有 1 个函数，执行了 1 个；有 5 行代码，执行了 5 行。

另外，这个命令还生成了一个子目录 coverage，其中的 coverage.json 文件是覆盖率的原始数据，lcov-report 目录包含了可以在浏览器打开的报告信息。在浏览器中打开文件 “coverage/lcov-report/index.html”，展示结果如图 12.8 所示。

从完美主义来说，覆盖率达到 100% 才算是合格代码，但实际开发中很难做到 100% 的覆盖率，这就要求开发者自己设置一个达标的门槛。

100% Statements 5/5		50% Branches 2/4		100% Functions 1/1		100% Lines 5/5			
File	Statements	Branches	Functions	Lines					
istanbul/	100%	5/5	50%	2/4	100%	1/1	100%	5/5	

图 12.8 lcov-report 文件展示效果

Istanbul 设置门槛的命令是“istanbul check-coverage”，用法如下：

```
istanbul check-coverage --statement 80
```

这句命令的意思是设置覆盖率门槛为 80%，如果实际覆盖率低于这个阈值，则报错。

Istanbul 还支持直接设置没有覆盖到的语句数量，命令如下：

```
istanbul check-coverage --statement -2
```

上面这句命令表示绝对值门槛，只要有超过 2 条语句没有被执行，测试即不通过。

除了设置语句数量，还可以设置分支和函数的数量或百分比，命令如下：

```
istanbul check-coverage --branch 90 // 分支
istanbul check-coverage --function 100 // 函数
```

这些参数完全可以结合起来使用，以设置一个符合开发者预期的综合门槛。当这些参数结合使用时，参数之间是“与”的关系，只要有一个条件没有达标，测试就会报错。

Istanbul 还提供了一套机制用于忽略那些不希望计入覆盖率的代码，语法如下：

```
/* istanbul ignore <word>[non-word] [optional-docs] */
```

这套注释语法有两个关键点：

- 必须放在多行注释语句中。
- 必须以“istanbul ignore”开头。

来看一个具体的案例，代码如下：

```
/* istanbul ignore if */
if(1 > 2){
  console.log(false);
}
```

代码中的 if 代码块，不管是否会执行，在计算覆盖率时都会被忽略。

这套注释语法关键字除了“if”，还包括“else”和“next”。使用“next”时表示接下来的那句代码将被忽略计算。

提示：Istanbul 的有关注释语法的更多介绍，请参考在线官方文档 <https://github.com/gotwarlost/istanbul/blob/master/ignoring-code-for-coverage.md>。

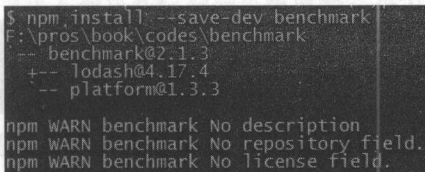
12.6 使用 Benchmark.js 进行基准测试

基准测试是测量和评估软件性能指标的活动。在前端开发中，性能调优是一项艰巨的任务，同样功能的不同实现、不同浏览器的特性都决定了代码执行速度的不同。那么，除了测试一个功能函数的执行速度外，比较不同代码的执行速度也是一项关键的工作。这就要求开发者对编写的 JavaScript 代码做一些基准测试，通过具体的数据来分析代码性能，以便做出更好的优化。Benchmark.js 正是帮助开发者完成基准测试的工具。

Benchmark.js 可以通过 NPM 包管理工具来安装，命令如下：

```
npm install --save-dev benchmark
```

运行结果如图 12.9 所示。



```

$ npm install --save-dev benchmark
F:\pros\book\codes\benchmark
-- benchmark@2.1.3
  +-- lodash@4.17.4
  -- platform@1.3.3

npm WARN benchmark No description
npm WARN benchmark No repository field.
npm WARN benchmark No license field.
  
```

图 12.9 安装 Benchmark.js

提示：Benchmark.js 依赖两个组件 lodash 和 platform。有关 lodash 和 platform 的更多信息，可以参考其官方网站 <https://lodash.com/> 和 <https://github.com/bestiejs/platform.js>。

假如某个项目需要测试正则表达式和字符串 indexOf 函数的性能，使用 Benchmark.js 编写的测试代码如下：

```

var Benchmark = require('benchmark');           // 引用 benchmark 组件
var suite = new Benchmark.Suite;                // 声明一个测试实例
suite.add('RegExp#test', function() {           // 添加正则的测试用例
  /o/.test('Hello World!');
});
suite.add('String#indexOf', function() {         // 添加 indexOf 的测试用例
  
```



```
'Hello World!'.indexOf('o') > -1;
});
suite.on('complete', function() {           // 添加一个事件监听：在测试完成后输出最快的用例名
  console.log('Fastest is ' + this.filter('fastest').map('name'));
});
suite.run();                                // 运行测试
```

将这段代码保存到文件 `index.js` 中，然后在控制台输入以下命令：

```
node index.js
```

过几秒钟之后，测试程序会在控制台输出测试结果，如图 12.10 所示。

```
$ node index.js
Fastest is String#indexOf
```

图 12.10 测试程序运行结果

如果想知道每一个测试用例的执行结果，可以监听“cycle”事件，代码如下：

```
suite.on('cycle', function(event) {
  console.log(String(event.target));
});
```

添加“cycle”事件后的运行结果如图 12.11 所示。

```
$ node index.js
RegExp#test x 8,645,238 ops/sec ±0.43% (91 runs sampled)
String#indexOf x 16,276,950 ops/sec ±0.27% (89 runs sampled)
Fastest is String#indexOf
```

图 12.11 添加“cycle”事件后的运行结果

除了 `cycle` 和 `complete` 事件之外，`Benchmark.js` 还支持很多其他事件，如下所示。

- `start`: 在测试程序运行开始的时候发生。
- `abort`: 在测试程序被取消的时候发生。
- `error`: 在测试程序出现错误的时候发生。
- `reset`: 在测试程序被重置的时候发生。

`Benchmark.js` 测试程序有两种创建方式，在本节的实例中使用的是基本创建方式，除了这种方式，`Benchmark.js` 还支持通过传入名称和配置的方式创建，代码如下：

```
var suite = new Benchmark.Suite('foo', {
  'onStart': onStart,
  'onCycle': onCycle,
  'onAbort': onAbort,
```

```
'onError': onError,
'onReset': onReset,
'onComplete': onComplete
});
```

Benchmark.js 不仅限于使用在 Node.js 开发环境，还支持直接在浏览器中使用。通过本节开始的安装结果已经了解到，Benchmark.js 依赖两个组件：lodash 和 platform。那么在浏览器中同样需要引用以下三个 JavaScript 文件，才能够运行 Benchmark.js 测试程序，代码如下：

```
<script src="lodash.js"></script>
<script src="platform.js"></script>
<script src="benchmark.js"></script>
```

接下来的测试代码写法与 Node.js 环境相同。

提示：Benchmark.js 的支持情况包括 Chrome 46+、Firefox 42+、IE 9-11、Edge 13、Safari 8+，以及 Node.js 0.10.x、0.12.x、4.x、5.x 和 PhantomJS 1.9.8。

12.7 实战演练：React 版备忘录项目的完整单元测试

前几节介绍了如何编写单元测试和检测测试覆盖率，本节采用一个实际的例子来介绍 React 应用的单元测试。本实例采用了之前章节介绍的 React 版备忘录应用。

实例选择了 Mocha 测试框架，该框架能运行在 Node.js 环境中，这样可以将单元测试和构建集成在一起。React 官方提供了单元测试组件“react-addons-test-utils”。首先安装必备的组件，命令如下：

```
npm i mocha chai babel-istanbul jsdom react-addons-test-utils -D
```

React 提供了两种渲染方式的单元测试，如下所示。

- **Shallow Rendering**：虚拟 DOM，将组件渲染成虚拟 DOM，但只渲染第一层，所以处理速度非常快。
- **DOM Rendering**：真实 DOM，将组件渲染到真实的 DOM 中，但需要 DOM 环境的支持。

在本实例中，采用 Shallow Rendering 测试框架，在 Node.js 中执行，需要模拟一些浏览器的对象，可以采用 jsdom 类库来模拟 DOM 对象。将模拟的“window”、“document”、“navigator”对象挂在全局对象中。本实例采用了 React 框架，还需要将 React 挂在全局对象上。在应用中还采用了 localStorage 对象，这里也需要模拟本实例中使用的相关 API。

提示：jsdom 是 WHATWG DOM 和 HTML 标准的 JavaScript 实现，主要被用于 Node.js，其官网地址为 <https://github.com/tmpvar/jsdom>。

在 test 目录下创建 setup.js 文件，在该文件中模拟上述需要的环境，文件的代码如下：

```
const react = require("react")
const jsdom = require("jsdom")
let items = {}
if (typeof document === 'undefined') {
  global.document = jsdom.jsdom('<!doctype html><html><body></body></html>');// 模拟 DOM
  global.window = document.defaultView; // 模拟 window
  global.navigator = global.window.navigator; // 模拟 navigator
  global.React = react // 全局 React 对象
  global.localStorage = { // 模拟 localStorage
    setItem(key, data) {
      items[key, data]
    },
    getItem(key) {
      return items[key]
    }
  }
  global.window.localStorage = global.localStorage
}
```

在执行 Mocha 测试的时候，将该文件采用 require 命令加入，命令如下：

```
mocha --compilers js:babel-core/register --require ./test/setup.js
```

这样在 Mocha 执行加载每一个单元测试文件时，都会自动加载这个文件。由于 React 采用了 JSX 语法，需要采用“:babel-core/register”组件，在运行的时候先编译 JSX 代码。

介绍一个具体的组件单元测试实例，组件的代码如下：

```
const Memo = props => {
  let memo = props.memo;
  // 根据备忘的状态设置样式
  let classNames = 'todo-item' + (memo.done ? ' done' : '');
  return (
    <div className={classNames}>
      <input type="checkbox" checked={memo.done}
        onChange={() => props.onToggleState(memo)} />
      <input className="color" type="color" value={memo.color}
        onChange={(e) => props.onChangeColor(memo, e.target.value)} />
    </div>
  )
}
```

```

    <span className="text">{memo.text}</span>
    <span className="pull-right del"
      onClick={() => props.onDelete(memo)}>X</span>
    <a className="pull-right">{new Date(memo.time).toLocaleString()}</a>
  </div>
);
}

```

首先测试渲染是否正确，可以采用 TestUtil 的 renderIntoDocument 方法渲染真实的 DOM。检测上述组件渲染的单元测试，代码如下：

```

describe("render", () => {
  it("render with done", () => {
    let data = { done: true } // 给定 done 为 true 的数据
    let app = testUtil.renderIntoDocument(<div><Memo memo={data} /></div>)
    let item = app.querySelector(".todo-item") // 找到渲染的元素
    should.exist(item, 'item rendered') // 检查元素是否被渲染
    should.equal(item.className, 'todo-item done', 'the done class has added')
  })
  ... // done 为 false 的分支测试
})

```

在上述测试代码，对 React 渲染的 DOM 做了检查，在这个 React 组件中，根据传入的状态来决定是否在 item 上添加“done”样式类。上面的代码通过传入不同的 Props，然后检测渲染的 DOM 对象的一些特征，来确定渲染是否正确。在这里还可以检测 Checkbox 的值，来确定 Checkbox 绑定的字段是否渲染正确。

在这个 React 组件中，绑定了一些鼠标事件。“react-addons-test-utils”组件提供了模拟事件的方法。采用该组件的 Simulate 对象，在该组件中，处理鼠标的事件由 Props 传入，因此在处理单元测试时，需要模拟一个事件处理函数来还原组件的真实使用情况，辅助单元测试。如组件中的“onToggleState”函数回调，可以采用如下的代码模拟：

```

const onToggleState = (data) => {
  data.done = !data.done
}

```

此时，可以针对这个回调函数来编写单元测试，代码如下：

```

const memo = { done: false } // 定义一个初始数据，便于渲染
this.app = testUtil.renderIntoDocument(<div><Memo onToggleState={onToggleState}
onChangeColor={onChangeColor} onDelete={onDelete} memo={memo} /></div>)
it("toggleState", () => {

```



```

let state = this.app.querySelector("input[type=checkbox]")
should.exist(state, 'the element has rendered')           // 检测 checkbox 是否存在
state.checked = true                                       // 设置 checkbox 为选中
testUtil.Simulate.change(state)                           // 模拟 change 事件
should.equal(memo.done, true, 'the state has changed')    // 通过检测数据来判断事件

```

执行的结果

```

  })
})

```

上述例子, 通过先定义一个初始数据对象, 将这个数据对象传递给组件完成渲染, 在模拟事件处理的函数中, 修改对象的字段。最后, 检测这个字段的值是否符合预期, 以此验证事件是否被成功执行。

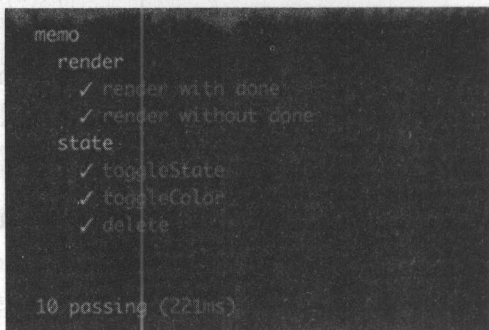
在 `package.json` 中, 可以将单元测试的运行脚本加入到 `scripts` 属性中, 代码如下:

```

"scripts": {
  "test": "mocha --compilers js:babel-core/register --require ./test/setup.js"
}

```

直接执行 “`npm run test`” 或者 “`npm test`” 命令运行单元测试。执行命令之后, 查看单元测试的结果, 如图 12.12 所示。



```

memo
  render
    ✓ render with done
    ✓ render without done
  state
    ✓ toggleState
    ✓ toggleColor
    ✓ delete

10 passing (221ms)

```

图 12.12 单元测试运行效果

在本章的前面介绍了如何采用 Istanbul 来检验单元测试的覆盖率。由于在本实例中, 采用了 JSX, 并且使用 Babel 编译, 运行 Istanbul 的命令比较复杂, 可以在 `package.json` 中的 `scripts` 属性中增加如下字段:

```

"scripts": {
  "cover": "node_modules/.bin/babel-node node_modules/.bin/babel-istanbul cover _mocha --require ./test/setup.js"
}

```

运行命令“npm run cover”之后，可以看到覆盖率的结果，如图 12.13 所示。

```
memo
render
  ✓ render with done
  ✓ render without done
state
  ✓ toggleState
  ✓ toggleColor
  ✓ delete

10 passing (180ms)

Writing coverage object [redacted]coverage/coverage.json]
Writing coverage reports at [redacted]coverage]

Coverage summary
Statements : 82.43% ( 61/74 )
Branches   : 66.67% ( 8/12 )
Functions  : 84.21% ( 16/19 )
Lines      : 85.29% ( 58/68 )
```

图 12.13 覆盖率测试效果

本节通过对一个 React 组件的单元测试，介绍了如何使用 Mocha 编写 React 组件的单元测试，以及如何使用 Istanbul 来检测单元测试的覆盖率。限于篇幅，本实例仅仅介绍了单个组件的单元测试，完整的代码请参考随书源码。

12.8 本章小结

Web 前端发展至今，已与从前的开发模式发生了巨大的变化。尤其在移动端，大量的前端实现采用 SPA 方案，前端的工程变得越来越复杂，但随着软件工程思路的更多引入，模块与模块之间功能更加独立和清晰，这给单元测试在真实业务项目上的落地提供了良好的环境。本章中介绍的 Jasmine、Mocha、Chai、Sinonjs Mocks、Karma、Istanbul、Benchmark.js 测试框架均为目前社区中的优秀作品，读者可以结合第 12.7 节的阅读完成对提及框架的学习使用。

13

第 13 章

前端性能优化实战

性能卓越的网站，往往可以给访问者留下良好的印象，因此更容易积累庞大的用户和受众群体，这是关乎网站成败的关键因素之一。本章将从前端开发者的角度出发，审视和探讨一些基本的网站性能优化思想，以及其在实战中的应用。

13.1 常用网站性能优化指标

对于网站性能指标最直接的印象可能是网站的响应速度，因为这是访问者最直观真实的体验。网站访问的过程从用户输入网站域名开始，通过 DNS 解析找到目标服务器，目标服务器收到请求后执行服务器及数据库等一系列操作，并将响应数据经过互联网发送到用户浏览器中，最终由浏览器处理响应数据并完成网页的渲染呈现。

13.1.1 网页的资源请求与加载阶段

下面打开 Chrome 浏览器的 DevTools, 看一看用户访问网站发送资源请求的全过程, 如图 13.1 所示。

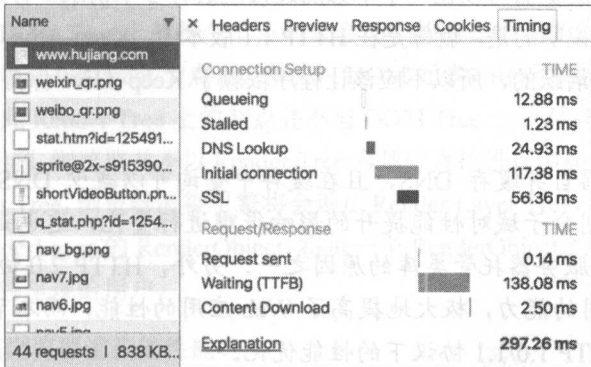


图 13.1 网站资源访问过程

从图中可以看到, 网页的访问请求过程, 包含了建立连接和请求响应两个阶段。由于 HTTP 协议是建立在 TCP 协议上的应用层协议, 上述浏览器中建立连接阶段, 即客户端到服务器之间建立 TCP 连接。

在建立连接的阶段, Queueing 和 Stalled 表示请求队列以及请求等待的时间。由于在 HTTP1 协议上, Chrome 浏览器只允许每个源拥有 6 个 TCP 连接, 因此如果请求处于正在排队或者等待状态, 说明请求需要等待以释放不可用的 TCP 套接字, 或者由于该请求的优先级低于关键资源而被渲染引擎推迟加载。DNS Lookup 则是执行 DNS 查询所用的时间。页面上的每一个新域都需要完整的往返才能执行 DNS 查询。Initial Connection 和 SSL 则包括 TCP 握手重试和协商 SSL 以及 SSL 握手的时间。

在请求响应阶段, Request sent 是发出网络请求所用的时间, 通常不会超过 1ms。Waiting (TTFB) 是等待初始响应所用的时间, 也称为等待返回首个字节的时间, 该时间将捕捉到服务器往返的延迟时间, 以及等待服务器传送响应所用的时间。Content Download 则是从服务器接收数据的时间。

由此可见, 为了实现网站的快速响应, 首先需要考虑的就是减少资源访问及加载阶段所消耗的时间。前面已经说过, 在使用 HTTP 1.0/1.1 协议时, Chrome 会将每个主机强制设置为最多 6 个 TCP 连接, 因此可以通过划分子域的方式, 将多个资源分布在不同子域上用来减少请求队列的等待时间。然而, 划分子域并不是一劳永逸的方式, 多个子域意味着更多的 DNS 查询时间。通常把域名拆分为 3 到 5 个比较合适。

另一方面，HTTP 是一个无状态的面向连接的协议，即每个 HTTP 请求都是独立的。然而无状态并不代表 HTTP 不能保持 TCP 连接，Keep-Alive 正是 HTTP 协议中保持 TCP 连接非常重要的一个属性。在 HTTP 1.1 协议中，Keep-Alive 默认打开，使得通信双方在完成一次通信后仍然保持一定时长的连接，因此浏览器可以在一个单独的连接上进行多个请求，有效地降低建立 TCP 请求所消耗的时间。但需要说明的是，就算是在 HTTP 1.1 版本中，Keep-Alive 也不能保证浏览器和服务端之间的连接一定是活跃的，所以不应该让程序依赖于 Keep-Alive 保持连接的特性，否则会有意想不到的后果。

注意：由于浏览器自身缓存 DNS，且在缓存中查询可以减少 DNS 的查询时间，所以实际应用中划分子域对性能提升的影响很难进行量化，这也是提倡在前端领域使用公共 CDN 服务器托管类库的原因之一。另外，HTTP 2.0 协议提供了单个 TCP 连接多路复用的能力，极大地提高了 Web 应用的性能，而本节所讲述的仍然是基于传统的 HTTP 1.0/1.1 协议下的性能优化。

接下来，在请求响应阶段，TTFB（Time To First Byte）所消耗的时间，更多体现在服务器程序的处理能力上，通常认为 TTFB 时间不应该超过 200ms，可以使用 CDN 加速以减少客户端到服务器的网络距离，也可以设置服务器缓存并通过为文件头设置 Expires 或 Cache-Control 来控制缓存，达到减少 TTFB 时间的目的，在处理访问静态资源的性能优化上，上述方法尤其重要。而 Content Download，在现今的网络条件下，通常不会成为性能瓶颈，因为如果发现 Content Download 时间过长，解决办法是减少发送的字节数。

13.1.2 网页渲染阶段

实际上，在 13.1.1 节的请求响应过程中，仅仅完成了网页的资源请求与加载。站在前端开发性能优化的角度，网站响应速度的快慢，不仅限于网站资源加载的速度，网页的渲染速度和前端 JavaScript 的运行速度同样直接影响着访问者的用户体验以及网站的整体性能。这一点在重度交互的网站应用中，体现得愈发充分。再次打开 Chrome 浏览器 DevTools 看一下在页面渲染上的性能消耗，如图 13.2 所示。

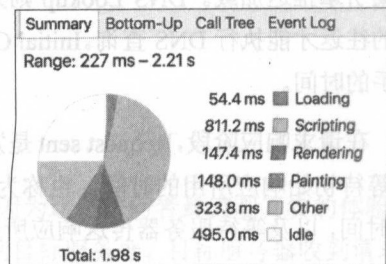


图 13.2 网页访问各阶段时间消耗

图中的 Rendering 以及 Painting 时间，就是浏览器在页面渲染过程中的时间消耗。在对浏览器渲染过程进行优化之前，再次了解一下浏览器是如何渲染网页的。

首先, 浏览器将从服务器获取的 HTML 文档构建成文档对象模型 DOM (Document Object Model), 与此同时浏览器会下载文档中引用的 CSS 与 JavaScript 文件。CSS 经过解析构成层叠样式表模型 CSSOM (CSS Object Model), 而 JavaScript 则会交给 JavaScript 引擎执行。紧接着, 文档对象模型 DOM 与层叠样式表模型 CSSOM 将构建渲染树 (Render Tree), Render Tree 上的节点称之为 RenderObject。DOM Tree 上的每个节点对象会递归检查是否需要创建 RenderObject, 并根据 DOM 节点类型创建 RenderObject 节点, 动态加入 DOM 元素。由于页面上的非可见元素并不会形成 RenderObject, 因此 Render Tree 上的节点并不与 DOM Tree 一一对应。为了方便处理定位、Z 轴排序、页内滚动等问题, 浏览器并不以 Render Tree 为基础直接进行渲染, 而是依据 RenderObject 生成新的 Render Layout Tree, 浏览器渲染引擎将会遍历 Render Layer Tree, 访问每一个 RenderLayer, 再遍历从属于这个 RenderLayer 的 RenderObject, 将每一个 RenderObject 绘制出来, 执行 Composite 合并 RenderLayer 并最终呈现给用户。

那么如何才能提高浏览器渲染的性能?

首先需要注意的是, 浏览器在加载到 JavaScript 节点后, 会交由 JavaScript 引擎执行, 如果 JavaScript 对 DOM 树进行读写操作, 则会影响 DOM 树的构建, 从而阻塞浏览器的渲染过程。解决此问题通常的做法是将 JavaScript 放在页面底部, 或者通过异步的方式加载 Javascript。另外, 过于复杂的 CSS 嵌套规则, 会影响 CSSOM 的生成, 导致渲染的时间被延长。在浏览器首次渲染以后, 页面上的元素还可能不断地被重新布局和绘制。对 DOM 节点进行添加和删除操作、通过 JavaScript 修改一些 CSS 属性、修改元素的 display 样式属性、移动或对节点添加动画、滚动或调整浏览器窗口大小等这些操作都会导致浏览器的重新布局以及重新绘制, 而修改元素的 visibility、背景色和前景色等也会导致重新绘制。如果处理不当, 这些动作可能会产生性能问题, 产生不好的用户体验。如图 13.3 所示, 诠释了浏览器重新布局以及重新绘制的过程。

Summary Bottom-Up Call Tree Event Log				
Filter		All	<input type="checkbox"/> Loading <input type="checkbox"/> Scripting <input checked="" type="checkbox"/> Rendering <input checked="" type="checkbox"/> Painting	
Start Time	Self Time	Total Time	Activity	
772.5 ms	12.0 ms	12.0 ms	Recalculate Style	
784.6 ms	13.5 ms	13.5 ms	Layout	
798.2 ms	2.9 ms	2.9 ms	Update Layer Tree	
801.1 ms	1.9 ms	1.9 ms	Paint	
803.2 ms	0.1 ms	0.1 ms	Paint	
808.2 ms	0.7 ms	0.7 ms	Composite Layers	
818.4 ms	0.0 ms	0.0 ms	Update Layer Tree	
850.3 ms	21.8 ms	21.8 ms	Recalculate Style	
872.1 ms	0.8 ms	0.8 ms	Update Layer Tree	
873.5 ms	16.0 ms	16.0 ms	Composite Layers	

图 13.3 浏览器在重新布局以及重新绘制过程中的性能消耗

在上述过程中, 最需要避免的是过多的重新布局, 重新布局必然会导致后续的重新绘制以及

合并。然而有一些特别的属性比如 `opacity` 或者 `transform` 可以在不同的层中单独绘制。对这种属性的访问，并不会导致重新绘制，而是直接执行了合并图层 `Composite Layouts`。由于图层合并 `Composite Layouts` 这个过程一般发生在 GPU 硬件渲染中，通常会非常高效，并可以减少渲染过程中的性能损耗。

13.1.3 JavaScript 脚本的执行速度

最后，JavaScript 脚本的执行速度是一个涵盖知识点非常多的话题。前面的章节已经介绍了 JavaScript 的性能测试工具 `Benchmark`。读者可以通过 `jsPerf` 提供的基于 `Benchmark` 运行的共享测试用例，来了解并比较不同 JavaScript 代码段的性能和执行速度，本节就不做过多赘述了。

注意：Nicholas C. Zakas 是公认的 JavaScript 专家，其著有前端经典著作《JavaScript 高级编程》以及《高性能 JavaScript》，读者可以通过阅读上述书籍来了解更多关于 JavaScript 性能方面的知识。

13.2 依旧有效的 Yahoo 性能优化法则

互联网时代的先行者雅虎！针对如何提高 Web 性能提出了许多优化建议，这些优化建议被广大开发者所采纳，并被称为“雅虎法则”或“雅虎军规”。时至今日，雅虎法则已经从最初的 14 个性能优化点发展为包含有内容优化、服务器、Cookie、CSS、JavaScript、图片和移动应用 7 大分类共计 35 条优化建议的一整套提高 Web 运行性能的最佳实践。

接下来花点时间了解一下雅虎法则中关于前端性能优化的几个要点。

- 减少 HTTP 请求

这是最为重要的一条优化法则。在上一节中，我们已经详细探讨了浏览器资源请求的过程。由于在 HTTP 1.X 协议的基础上，TCP 连接的多路复用与资源的并行下载存在诸多限制，减少 HTTP 请求可以最大程度地降低资源请求阶段的性能消耗，从而提升网站访问速度。从另一个角度来说，减少 HTTP 请求也可以降低服务器负载，减少并发并提升服务器的处理能力。减少 HTTP 请求的手段有很多，比如合并 CSS 与 JavaScript 文件、使用 CSS Sprites、内联图像“`data:URL scheme`”等。

- 压缩 CSS 和 JavaScript 代码

压缩 CSS 与 JavaScript 代码的体积，精简不必要的空格、换行、缩进等。代码的字节数减少，代码对应的下载时间也会随之减少。

- 去除重复引用的脚本

重复脚本在 Internet Explorer 浏览器中会导致多余的 HTTP 请求，也会带来不必要的运算，降低网站的性能。

- 可缓存的 AJAX

AJAX 经常被提及的一个好处就是异步性。该技术异步地从服务器传输信息，为用户带来即时的反馈。但使用 AJAX 并不保证用户不会等待异步请求，通过 Expire 或者 Cache-Control 头来实现缓存，可以从缓存中读取内容，提升性能。

- 延迟加载非必要脚本

页面上有些内容并不需要立刻加载。非首屏的图片资源，需要经过用户操作才能呈现的非可见元素等都可以进行推迟加载。

- 预加载

预加载看起来和延迟加载恰恰相反，实际上预加载是指在浏览器空闲阶段预先加载将来用户可能会访问到的内容，从而提高页面的即时响应能力，优化用户体验。

- 减少 DOM 元素数量

DOM 元素过多意味着需要加载更多的数据，在使用 JavaScript 遍历 DOM 时更加低效，也意味着执行布局和绘制时产生更多的性能损耗。

- 减少 DOM 访问次数

DOM 被认为天生即慢，通常需要将经常访问的 DOM 对象缓存，避免过多地使用 JavaScript 修改页面布局。

- 避免使用 iframe

即使引入页面内容为空，iframe 也需要消耗时间下载，会阻止页面加载，并且缺少语义。

- 优化图像

设计师提供的图片大多数都存在压缩空间，或者可以尝试将 GIF 格式转换为 PNG 格式。

- 优化 CSS Sprites

CSS Sprites 中，图片水平排列较之垂直排列体积更小，可以将颜色接近的组合在一起。不要在 Sprites 中留有较大的空隙。空隙虽然不会增加文件大小，但对于用户需要更多的内存来把图片解压为像素地图。

- 不要在 HTML 中缩放图片

在 HTML 中设置的图片长宽小于实际下载图片长宽，很显然实际加载的图片比需要的图片体积更大。

- 减少 Cookie 体积

Cookie 信息通过 HTTP 文件头在服务器和浏览器之间进行传递，因此保持 Cookie 尽可能小，可以减少用户的数据传输时间。

以上列举了雅虎法则中前端优化最常用的几条，更多的内容可以查阅雅虎开发者官方网站。通常一个网页从发起服务器请求到实现对最终用户的展现，在客户端所花费的时间大约会占到总时耗的 80%，因此客户端的性能优化显得尤为重要。这也是成为优秀前端工程师必须要掌握的基础知识。

注意：Steve Souders 是 Google Web 性能布道者和雅虎前首席性能工程师，在其著作《高性能网站建设指南》和《高性能网站建设进阶指南》中详细介绍了雅虎法则，并以此为基础，创建了网站性能分析工具 YSlow。最新的 YSlow 以雅虎法则中 23 个可测试的优化点作为分析依据，并已经支持目前多数主流的浏览器。

13.3 性能优化工具使用实战

在网站进行优化之前，首先需要进行性能分析。目前主流的性能分析工具，大多会提供浏览器插件以及在线分析等几种方式。下面就选取其中比较有代表性的性能优化分析工具做一些简单的介绍。

13.3.1 YSlow

上一节介绍了雅虎性能优化法则。YSlow 是依据雅虎法则中 23 条可测试的性能法则构建的网站性能分析工具，并提供了多种浏览器插件以及跨平台的支持，如图 13.4 所示。

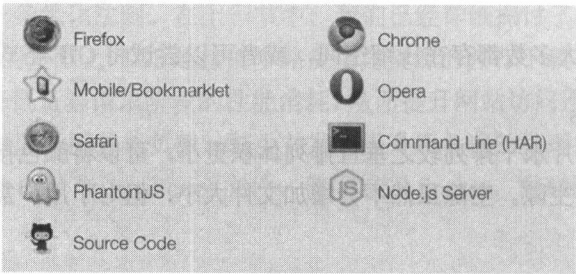


图 13.4 YSlow 的浏览器平台支持

以 Chrome 浏览器为例，首先访问 YSlow 官方网站 yslow.org，进行浏览器插件安装。安装完

毕以后，重启浏览器并打开需要进行性能优化分析的网页。单击 Chrome 右上角的 YSlow 标记之后，单击 YSlow 插件“Run Test”按钮，开始进行网站性能优化分析，等待分析完毕，如图 13.5 所示。

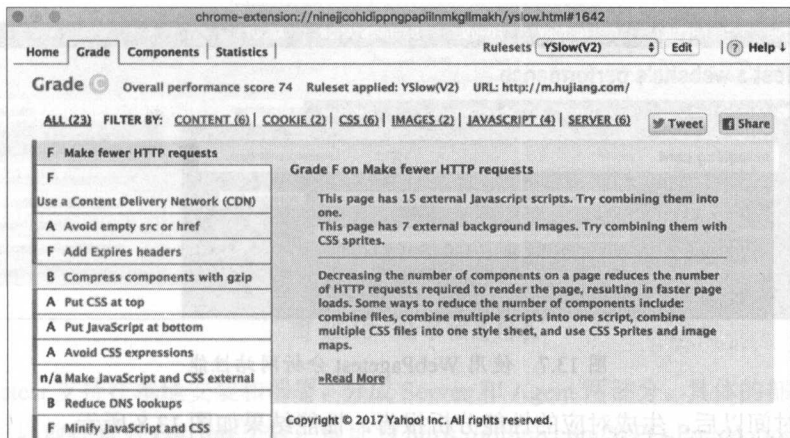


图 13.5 YSlow 性能分析结果

YSlow 会根据雅虎法则的优化建议对网页性能进行整体等级评价，待分析完成后，读者可以有针对性地处理 YSlow 提供的优化建议点。

13.3.2 PageSpeed

PageSpeed 是由 Google 公司提供的一款性能优化分析工具。访问 PageSpeed 官方网站 <https://developers.google.com/speed/pagespeed/>，只需要在输入框里输入需要分析的网页地址，就可以完成在线分析，并获取网页优化建议以及解决问题的方法，如图 13.6 所示。

提示：PageSpeed 还提供了 CLI (Command Line Interface) 工具 psi，使用 NPM 执行命令“npm install psi”进行安装，详情请参考 <https://www.npmjs.com/package/psi>。



图 13.6 PageSpeed 性能分析结果

13.3.3 WebPagetest

WebPagetest 是 Google 开源项目“Make the Web Faster”的子项目在线版本,官网地址为 <http://www.webpagetest.org/>。访问 WebPagetest 网站输入要分析的网页地址,如图 13.7 所示。

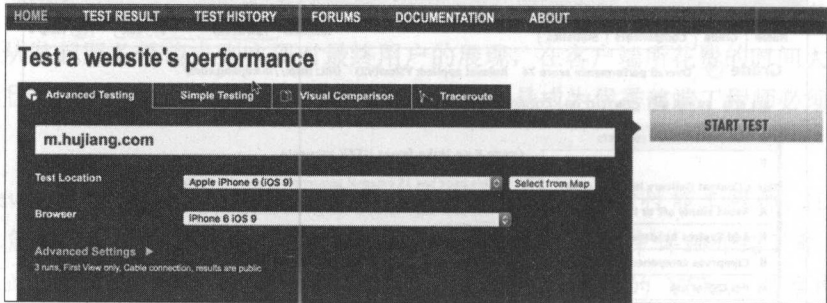


图 13.7 使用 WebPagetest 分析网站性能

等待一段时间以后,生成对应的性能分析报告,性能结果如图 13.8 所示。

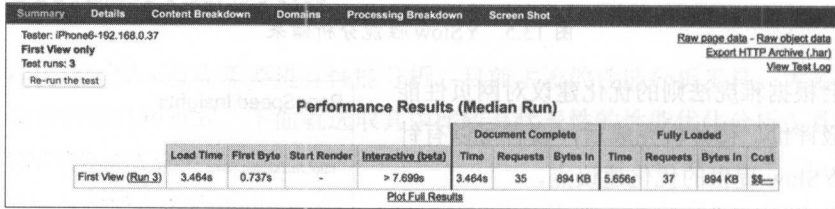


图 13.8 WebPagetest 生成性能结果

性能报告中的请求瀑布图,如图 13.9 所示。

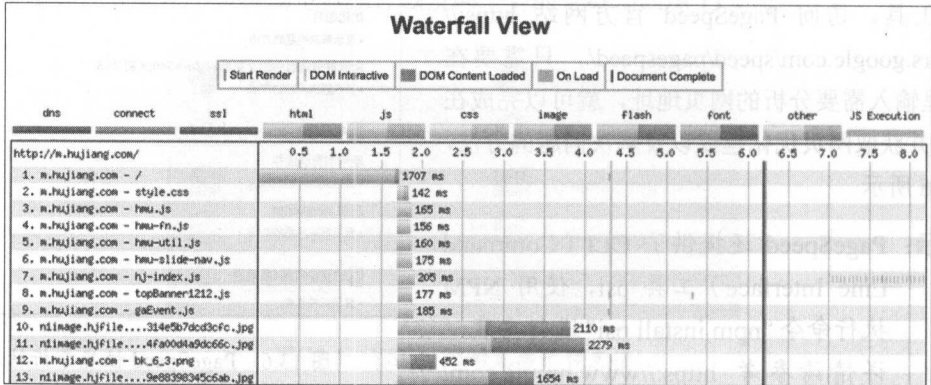


图 13.9 请求瀑布图

性能报告中的请求信息列表，如图 13.10 所示。

Request Details											
Request Details											
#	Resource	Content Type	Request Start	DNS Lookup	Initial Connection	SSL Negotiation	Time to First Byte	Content Download	Bytes Downloaded	Certificates	Error/Status Code
1	http://m.huiliang.com/	text/html; charset=utf-8	-	-	-	-	1666 ms	41 ms	27.0 KB	-	200 -
2	http://m.huiliang.com/_015/styles/style.css	text/css	1.693 s	-	-	-	77 ms	65 ms	3.4 KB	-	200 -
3	http://m.huiliang.com_w2015/scripts/hmu.js	application/x-javascript	1.702 s	-	-	-	163 ms	2 ms	0.7 KB	-	200 -
4	http://m.huiliang.com_15/scripts/hmu-fn.js	application/x-javascript	1.703 s	-	-	-	151 ms	5 ms	0.7 KB	-	200 -
5	http://m.huiliang.com/_scripts/hmu-vill.js	application/x-javascript	1.703 s	-	-	-	160 ms	-	2.2 KB	-	200 -
6	http://m.huiliang.com_pte/hmu-slide-nav.js	application/x-javascript	1.704 s	-	-	-	174 ms	1 ms	1.2 KB	-	200 -
7	http://m.huiliang.com/_scripts/hj-index.js	application/x-javascript	1.705 s	-	-	-	198 ms	7 ms	3.0 KB	-	200 -
8	http://m.huiliang.com_pte/ocBanner1212.js	application/x-javascript	1.705 s	-	-	-	176 ms	1 ms	0.8 KB	-	200 -

图 13.10 请求信息列表

WebPagetest 支持在本地安装和部署，分成 Server 和 Agent 两部分。具体的部署方案本文不再赘述，读者可以前往官方 GitHub 下载，地址为 <https://github.com/WPO-Foundation/webpagetest>。

13.4 HTTP 协议头缓存实战

提供更好的用户体验一直是 Web 开发者追求的目标。一个引用大量静态资源的 Web 页面，资源的加载速度是影响页面加载耗时的最大因素。很多固定静态资源，如图片、脚本、样式文件等若都可以通过缓存保存在客户端，不必每次都从服务器请求，将节省大量请求时间。那么，客户端和服务端通过怎样的方式，才能确定客户端知道是否需要使用缓存文件。本节将会介绍 HTTP 缓存的具体细节。

13.4.1 客户端缓存流程

现代浏览器会根据 HTTP 协议中的缓存，实现本地缓存功能。在请求一个新的文件时，浏览器发送 HTTP 请求到服务端。接到服务端的响应后，浏览器会将请求的资源储存在本地，留作以后使用，如图 13.11 所示。

服务端响应头中，会带有文件相关的缓存策略，告诉浏览器文件是否需要缓存以及缓存何时过期等信息。当浏览器再次请求文件时，会先判断缓存中是否有相应的文件以及是否过期，未过期则直接从缓存中读取文件，不会再向服务器发送请求，如图 13.12 所示。

图 13.11 初次请求

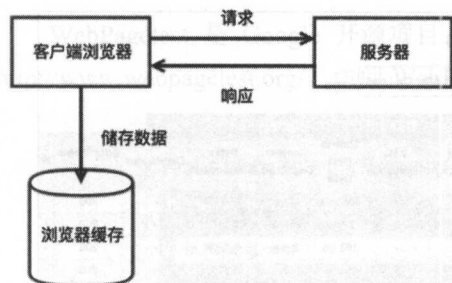


图 13.11 初次请求

浏览器本地是否有缓存并且可用?

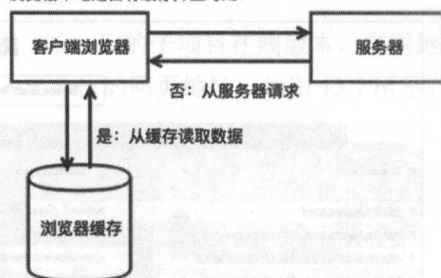


图 13.12 再次请求

13.4.2 缓存协议内容

HTTP 头中关于缓存相关的属性，主要有以下几种。

(1) Expires: 指定缓存过期的时间，是一个绝对时间，但受客户端和服务端时钟和时区差异的影响。

(2) Cache-Control: 比 Expires 策略更详细，优先级比 Expires 高，其值可以是以下五种情况。

- no-cache: 告诉客户端不使用缓存。
- no-store: 告诉客户端不要缓存响应。
- public: 缓存响应，并可以在多用户间共享。
- private: 缓存响应，但不能在用户间共享。
- max-age: 缓存在指定时间（单位为秒）后过期。

(3) Last-Modified/If-Modified-Since: 指定响应资源的最后修改时间。如果响应头中包含 Last-Modified，再次请求时通过 If-Modified-Since 将最后修改时间告诉服务端，服务端判断文件是否有过修改，再决定返回新内容还是通过 HTTP 状态码 304 告诉客户端使用缓存。

(4) ETag/If-None-Match: 区别资源内容的唯一标识，需要配合 Cache-Control 使用。当文件最后修改时间发生变化，但文件内容并无改变时，也应该使用缓存。如果响应头中包含 ETag，再次请求时通过 If-None-Match 将内容标识告诉服务端，服务端比较内容是否有改变后，再决定返回新内容还是通过 HTTP 状态码 304 告诉客户端使用缓存。

使用浏览器的调试工具或者通过代理工具 Fiddler 等，可以查看请求和响应的头部内容。在 Chrome 浏览器中，打开调试窗口切换至 Network 选项，可以看到客户端发送的请求列表。初次请

求时, 服务器返回状态码 200; 服务器判断文件未修改返回状态码 304; max-age 或 Expires 未过期时, 客户端直接读取本地缓存, 如图 13.13 所示。




Name	Status	Type	Initiator	Size
 cachedemo	200 OK	document	Other	478 B 158 B
 cachedemo	304 Not Modified	fetch	VM34249... Script	262 B 158 B
 cachedemo	200 OK	fetch	VM34249... Script	(from disk cache)

图 13.13 网络请求中的缓存状态

HTTP 头部中的缓存设置, 如图 13.14 所示。

▼ Response Headers	view source
Cache-Control: max-age=5	
Connection: keep-alive	
Date: Mon, 13 Mar 2017 05:29:28 GMT	
ETag: "9e-wQQswEu3yQAJPO8dNt0DjK2buVw"	
Expires: Wed Apr 12 2017 00:00:00 GMT+0800 (CST)	
Last-Modified: Sun Mar 12 2017 00:00:00 GMT+0800 (CST)	
▼ Request Headers	view source
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8	
Accept-Encoding: gzip, deflate, sdch, br	
Accept-Language: zh-CN,zh;q=0.8	
Cache-Control: max-age=0	
Connection: keep-alive	
Host: localhost:3005	
If-Modified-Since: Sun Mar 12 2017 00:00:00 GMT+0800 (CST)	
If-None-Match: "9e-wQQswEu3yQAJPO8dNt0DjK2buVw"	

图 13.14 HTTP 头部中的缓存设置

13.4.3 实战演练: HTTP 缓存

本节通过 Node.js 在本地启动一个服务器测试 HTTP 缓存功能。这里以“Cache-Control:max-age”为例, 为了方便测试缓存过期后的效果, 将 max-age 的值设为 5s, 服务端代码如下:

```
response.setHeader('Cache-Control', 'max-age: 5');
```

在 Chrome 中访问本地实例测试地址 `http://localhost:3005/cachedemo`, 接着 5s 内在控制台通过 Fetch API 再次请求, 5s 后再次发送请求。

控制台发送请求代码如下:

```
fetch('http://localhost:3005/cachedemo')
```

在 Nextwork 下观察三次网络请求, 如图 13.15 所示。




Name	Status	Type	Initiator	Size
 cachedemo	200 OK	document	Other	339 B 115 B
 cachedemo	200 OK	fetch	VM25298... Script	(from disk cache)
 cachedemo	200 OK	fetch	VM25298... Script	339 B 115 B

图 13.15 测试 max-age

从图中可以看到:

- 第一次请求时没有缓存, 因此直接从服务器访问资源;
- 第二次请求时, 浏览器本地已经存在缓存并且未过期, 因此直接从本地磁盘读取了缓存内容;
- 第三次 5 秒后的再次请求, 因为本地缓存已过期, 浏览器重新从服务器请求资源。

13.5 资源按需加载实战

13.5.1 基于 RequireJS 的按需加载

RequireJS 是一种 AMD (Asynchronous Module Definition 即异步模块定义) 的实现, 采用异步加载模块, 因此模块的加载不会影响后续代码的运行, 官网地址为 <http://requirejs.org/>。AMD 通过 `require` 函数加载模块, 接收两个参数, 实例代码如下:

```
require([module], callback);
```

- `module`: 所有需要加载的模块, 数组类型。
- `callback`: 模块加载并运行完后执行的回调函数。

在实际开发中, 开发者需要在页面上引入 RequireJS 的加载器代码和一个主 JavaScript 文件用来加载依赖模块, 假设主 JavaScript 文件叫作 `app.js`, 并需要引入 `a`、`b`、`c` 三个依赖模块, 在 `app.js` 文件中可以定义依赖模块的路径, 代码如下:

```
requirejs.config({                                // 配置
  baseUrl: 'js/modules',                         // 设置基础 Url, 后续模块不用重复写完整路径
  paths: {                                        // 每个模块的路径
    "module-a": "a.js",
    "module-b": "b.js",
    "module-c": "c.js"
  }
})
```

使用 `require` 函数引入三个依赖模块，代码如下：

```
// 可以通过回调函数的参数调用模块的能力
require(['module-a', 'module-b', 'module-c'], function(a, b, c) {
    // 具体业务代码
})
```

在每个模块的 JavaScript 文件中，开发者首先需要定义模块，AMD 规范提供了一个 `define` 函数，如果一个模块不依赖其他模块，可以直接进行定义，代码如下：

```
define(function () {
    var add = function (x, y) {
        return x+y;
    }
    return {
        add: add
    }
})
```

如果依赖其他模块则可以把依赖模块数组作为第一个参数传入，代码如下：

```
define(['module-c'], function(c) {});
```

RequireJS 可以帮助开发者异步加载 JavaScript 代码，解决了模块之间的依赖关系，提升应用的整体质量和性能。

13.5.2 基于 Webpack 的按需加载

CommonJS 规范虽然本身采用同步加载模块，但也提出了 `Modules/Async/A` 规范，定义了一套 `require.ensure` 用于处理异步加载。`require.ensure` 会将模块下载下来，Webpack 作为一个模块加载器同时也是一个打包工具，所以开发者不需要特意地去定义模块，Webpack 会使用“Code Splitting”技术实现分批打包和按需加载。假设依旧存在 a、b、c 三个 JavaScript 文件，在 `app.js` 文件中使用以下代码：

```
require('./module/a');
require.ensure(['./module/b'], function(require) {
    require('./module/c');
})
```

在最后的打包文件中，会有一个 `bundle.js` 和 `0.bundle.js`。`bundle.js` 包含 Webpack 信息、模块 a 和 `app.js` 的内容。`0.bundle.js` 包含模块 b 和模块 c 的内容。最后开发者只需要在页面中引入 `bundle.js` 文件，则 `0.bundle.js` 文件会被动态引入。

虽然 b.js 和 c.js 被打包在了一个文件, 但只有 c.js 的内容被执行, 如果需要运行 b.js, 仍需要手动输入 “require('./module/b')”, 这种模式虽然需要手动运行模块, 但是可以控制依赖模块的执行顺序, 而在 RequireJS 中, 依赖模块的加载和运行是不固定的。

提示: Modules/Async/A 规范, 参考地址为 <http://wiki.commonjs.org/wiki/Modules/Async/A>。

13.5.3 图片懒加载

前端页面加载中另一个对用户体验很大的因素是图片, 图片往往是页面上体积最大的资源。当页面中存在太多图片时, 页面加载时间较长。懒加载的原理是通过监听页面滚动事件, 当图片进入可视区域时, 再进行图片的加载。

要实现这个技术, 首先在 IMG 元素的自定义属性中存放图片的真实路径, 再动态地赋值给 IMG 标签的 src 属性, 这里将 IMG 的自定义属性命名为 “data-src”, 代码如下:

```

```

这里的 “data-src” 属性存储的就是真实路径, 在页面渲染时, 因为浏览器不会处理该自定义属性, 所以不会自动加载图片。接着, 监听页面的滚动事件, 代码如下:

```
window.addEventListener('scroll', loadImage, false)
```

在 loadImage 函数中需要决定哪些图片被加载, 并将 “data-src” 中的真实地址赋值给执行下载的 src 属性, 代码如下:

```
var imgList = Array.prototype.slice.call(document.querySelectorAll('img')); // 获取所有图片
function loadImage() {
    for (var i = 0; i <= imgList.length; i++) { // 遍历图片集合
        var el = imgList[i];
        if (isShow(el)) { // 如果图片出现
            el.src = el.getAttribute('data-src'); // 给 src 赋值
            imgList.splice(i, 1); // 去除已加载的
        }
    }
}
function isShow(el) {
    // 获取图片在屏幕中图片顶部与屏幕顶部的距离、图片底部与屏幕顶部的距离、
    // 图片左边框与屏幕左侧的距离、图片右边框与屏幕左侧的距离, 及图片的宽高
    var rect = el.getBoundingClientRect();
    // 如果 rect.right 或 rect.bottom 小于 0, 意味着图片整体未进入屏幕
    // 如果在大于 0 的情况下, rect.top 和 rect.left 比较 innerWidth 和 innerHeight
```

```
// 保证图片在部分进入后就开始加载
var isAppear = rect.right > 0 && rect.left < window.innerWidth &&
    rect.bottom > 0 && rect.top < window.innerHeight
return isAppear;
}
```

这里提供了一个最简单的图片懒加载方案。真实的业务场景中还需要考虑用户下拉速度、页面高度的固定性、iScroll 等第三方插件库的使用情况。笔者同时也推荐开发者使用一些开源的懒加载库，以方便业务开发。

提示：iScroll 是 Matteo Spinelli 开发的旨在解决移动端浏览器的区域滚动问题，使用原生 JavaScript 编写，不依赖于任何第三方框架，最新版本为 iScroll5。官网地址为 <http://iscrolljs.com/>。

13.6 不同网络类型的优化实战

13.6.1 获取网络类型

Web APIs 在 Navigator 接口中新增 connection 属性来获取网络状态。开发者可以通过 navigator.connection.type 获取网络类型，包括 unknown、Ethere、WIFI、2G、3G、4G、none。理想情况下可以利用这个信息对资源进行优化，但是这个 API 仍属于一个实验性质的属性，支持浏览器的情况如图 13.16 所示。

Network Information API - UNOFF

Global 25.69% + 4.26% = 29.94%

The Network Information API enables web applications to access information about the network connection in use by the device.

Current aligned Usage relative Date relative Show all

IE	Edge	Firefox	Chrome	Safari	Opera	iOS Safari	Opera Mini	Android Browser	Chrome for Android
			49					4.4	
			55			9.3		4.4.4	
11	14	51	56	10	43	10.2	all	53	56
	15	52	57	10.1	44				
		53	58	TP	45				
		54	59						

图 13.16 Network Information API 浏览器支持情况

可以看出大部分浏览器都不能很好地支持该属性，所以通常需要开发者主动考虑用户的弱网情况，在代码层面上进行优化，或者通过混合开发模式在 APP 中通过接口获取用户的网络类型。

13.6.2 弱网图片优化

图片通常是一个页面上最消耗网络资源的内容。如今很多手机已经配上了高分辨率屏幕，因此在实际开发过程中，开发者会使用高分辨率图片缩放以保证清晰度。但这样做的同时也增大了图片资源的体积，所以通常在 Web 开发环境下使用两倍分辨率即可，在 APP 中甚至可以根据网络环境使用更低分辨率的图片。

在前端开发中，开发者会使用雪碧图来减少对资源的请求数量。这种方式正确与否要视具体情况而定。将高分辨率图片整合成一张雪碧图是不合理的，因为这样的图片会需要很长的加载时间，从而影响用户体验。正确的做法是仅将小图标整合到雪碧图，并控制每张雪碧图的体积，如果超过了上限，则整合第二张雪碧图。在雪碧图合成中要使用合理的排放策略，甚至手动合成，避免有一些图标在合成过程中产生大量留白，造成资源的浪费。

说明：雪碧图，即 CSS Sprite，将页面中的图标或者背景图片合并到一张大图中，通过控制 CSS 的 background-position 属性确定图片呈现的位置。

图片格式也是优化的重点。如果不需要透明图层，比如背景图片，则避免使用 PNG 格式。大尺寸图片需要选择 JPG 格式，同时加一定的压缩比例，这样能够把体积减少到原来 PNG 格式的几分之一。开发者还可以在 APP 中使用 WebP 格式的图片，兼顾实现效果和体积。

另外还可以使用字体图标。比如，需要产生一定动画效果的图标至少需要两张图片来实现，然而使用字体图标则可以通过 CSS 来装饰。同理，一些简单的 GIF 动画，如“loading”效果图也可以使用原生 CSS 实现，不但减少了网络请求数量，同时还提升了性能。

13.6.3 弱网缓存优化

页面端的数据请求也是一个对弱网敏感的优化点，尤其在单页应用中，可能会同时拉取多个接口的数据。在一些 2G 网络下，页面会长时间处于加载数据的状态。遇到这种情况，开发者可以选择在内存中缓存请求数据。例如，入口页为 A 页面，依赖数据源 a，则可以在代码中定义一套缓存，实例代码如下：

```
var store = {
  a : {
    url: '',           // 请求的地址
    type: 'GET',       // 请求的类型
    param: '',         // 请求的参数
    cache: {}          // 缓存请求的结果
  }
}
```

当首次访问 A 页面时,通过调用 `store.a` 获取对应 A 页面请求参数信息,在成功获取数据后进行数据缓存。当视图切换回到 A 页面时,先判断是否存在 `store.a` 对象的 `cache` 属性,如果存在数据则读取并返回,不进行网络请求。此外,还可以在应用中默认进行网络请求,设置请求的超时时间。在请求的 `complete` 事件中,假设发生请求超时,则读取缓存数据渲染页面,部分实例代码如下:

```
complete: function(XMLHttpRequest, status) {
    if (status == 'timeout') {
        var data = store.a;
        // 下面处理业务代码
    }
}
```

这种缓存策略需要注意的是,如果存在其他修改当前缓存的可能,则需要在发出请求成功后的回调函数中保证缓存的更新。

除了内存中的缓存策略以外,HTML 5 还提供了 `localStorage` 作为客户端缓存方案,`localStorage` 在现代浏览器中基本都得到了支持,如图 13.17 所示。

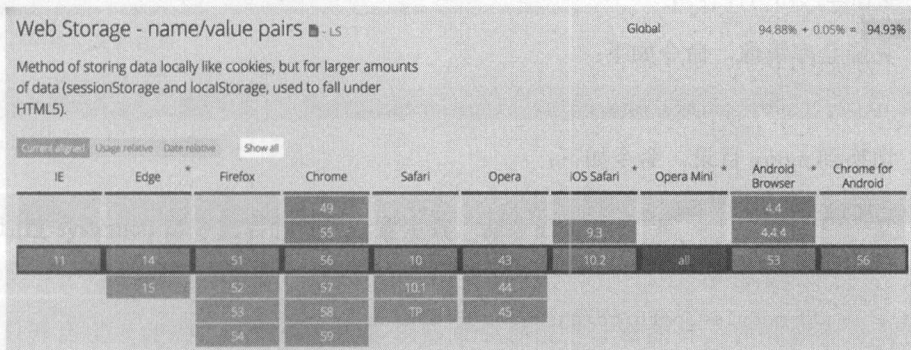


图 13.17 Web Storage 浏览器支持情况

注意: `localStorage` 缓存数据的存储类型为字符串格式,所以在存储对象字面量时需要先通过 `JSON.stringify` 方法处理转换,在之后的使用中调用 `JSON.parse` 重新转换为原对象数据结构。

13.7 实战演练: Nginx 配置 Combo 合并 HTTP 请求

Nginx 是一个开源高效的 HTTP 服务器,通过简单的配置能够提供丰富的功能,最重要的是,它对服务器的配置要求极低。Nginx 选用了由事件驱动的异步非阻塞模型,而不是传统的依赖多

线程来响应, 因此, 在同等资源的情况下能够提供极大的并发请求。对于前端开发者来说, 选用 Nginx 是一个主流并且高性价比的选择。

13.7.1 安装 Nginx 和文件合并模块

要玩转 Nginx, 首先推荐使用 Linux 操作系统的服务器来搭建服务, 一台普通的云服务器即可。文件合并模块选择开源插件 nginx-http-concat, 操作系统为 CentOS 7。一般情况下安装 Nginx 推荐添加 EPEL (全称 Extra Package for Enterprise Linux, 即企业级 Linux 附加包) 安装源, 然后通过 CentOS 包管理器 yum 安装。这种方式的安装简单且方便升级。因为安装模块需要重新编译 Nginx 安装包, 所以这里介绍通过二进制包来安装, 步骤如下。

(1) 下载最新的 Nginx 稳定版本二进制包, 以版本 1.10.3 为例, 命令如下:

```
wget http://nginx.org/download/nginx-1.10.3.tar.gz
```

(2) 解压缩安装包, 命令如下:

```
tar zvfz nginx-1.10.3.tar.gz
```

(3) 克隆仓库镜像, 命令如下:

```
git clone git@github.com:alibaba/nginx-http-concat.git
```

(4) 切换到 nginx 目录, 命令如下:

```
cd nginx-1.10.3
```

(5) 在构建参数中添加模块并构建, 代码如下:

```
./configure --add-module=/path/to/nginx-http-concat
make
make install
```

(6) 查找 Nginx 地址, 命令如下:

```
// 找到 Nginx 执行文件的地址, 可以配置命令行环境, 或者通过路径调用
whereis nginx
// 查找到 Nginx 的默认配置文件地址, 如果异常默认在/etc/nginx/conf.d/default.conf
nginx -t
```

通过 cat 指令读取配置文件, 可以看到 root 地址, 对应网站根目录。

(7) 启动 Nginx, 命令如下:

```
// 推荐使用 systemctl 来启动, 也可以通过 nginx 命令来启动
```

```
sudo systemctl start nginx
// 开机自启动, 可选
sudo systemctl enable nginx
```

13.7.2 配置 Nginx 和 Combo

nginx-http-concat 适用于 Nginx 的文件合并模块, 可以将多个对静态资源的 HTTP 请求合并为一个, 进而减少 HTTP 请求数。假设原来需要 a.js、b.js、c.js 三个文件, 产生三次请求, 通过该模块可以在一个请求中完成, 实例地址如下:

```
http://example.com/static/??a.js,b.js,c.js // 使用??作为标示表示合并文件
```

需要修改 Nginx 的配置, 文件配置如下:

```
location /static/ {
    # nginx-http-concat 主开关
    concat on
    # 最大合并文件数
    # concat_max_files 10;
    # 只允许同类型文件合并
    # concat_unique on
    # 允许合并的文件类型, 多个以逗号分隔。如: application/x-javascript, text/css
    # concat_types text/html
}
```

然后通过 systemctl 命令重启服务让配置生效, 命令如下:

```
sudo systemctl restart nginx
```

这里只开启了合并功能, 其他功能根据需求调整取消注释即可, 详细参数可以查看网址 <https://github.com/alibaba/nginx-http-concat>。

13.8 本章小结

Web 优化的黄金法则中指出对于访问一张网页只有 10%~20%的时间花在下载服务器端响应的 HTML 文件, 80%~90%的时间用在前端资源的下载和执行, 比如 CSS、JavaScript、Images 等, 前端性能优化的重要性不言而喻。互联网产品的迭代速度以“快”著称, 持续地保持站点快速响应和流畅交互, 给从事前端开发的人员带来了极大的挑战。本章节选了在移动端最常用的几种优化手段, 希望对读者开发出一款快速的应用有所帮助。

14

第 14 章

项目实战：搭建直播平台 (Cordova+Koa+React)

通过前面章节的学习，大家应该已经掌握了许多移动 Web 开发的知识和技巧。为了巩固所学知识，本章将带领大家从零开始搭建一个直播平台，并尽可能涵盖之前所有的知识点。

14.1 项目的安装和启动

本节将介绍项目的安装和启动。首先打开附属源代码中的 iKtalk 文件夹，文件目录如图 14.1 所示。

其中，client 文件夹存放了客户端代码，server 文件夹存放了服务端代码。客户端采用 React 框架编写，服务端采用基于 Node.js 的 Koa 框架编写。

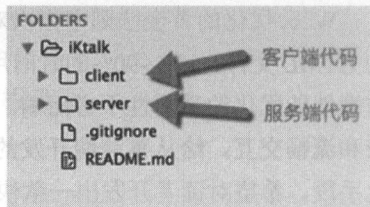


图 14.1 iKtalk 文件夹

提示：本章实例代码可以在 GitHub 下载，地址为 <https://github.com/ikcamp/iKtalk>。

14.1.1 安装依赖

打开终端，分别进入 client 和 server 文件夹，在文件夹根目录下输入如下命令来完成依赖安装：

```
npm install
```

注意：如果安装依赖的时候系统提示没有权限，请切换为 root 用户再安装。

14.1.2 启动项目

完成依赖安装后，在 server 文件夹根目录下启动服务，命令如下：

```
npm run dev
```

之后可以看到服务启动，并监听 4412 和 4413 端口。

在 client 文件夹根目录下编译静态文件并启动静态资源服务，命令如下：

```
npm start
```

静态资源服务启动后会自动打开系统默认浏览器。为了方便演示，此处建议读者把默认浏览器设置为 Chrome。在浏览器中可以看到直播平台的主界面，也就是直播列表页，如图 14.2 所示。

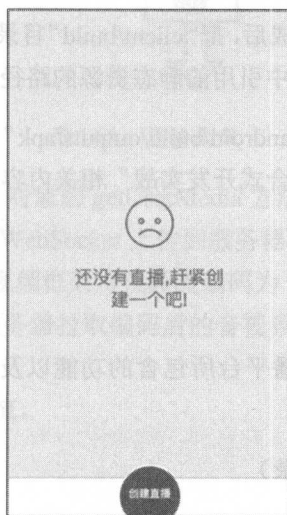


图 14.2 直播列表页

作为一种推荐的交互方式，在没有实质内容可供展示的时候，直播列表页会给用户一个“友

好”的提示。单击页面下方的“创建直播”按钮可创建一个直播间。

14.1.3 Cordova 打包

笔者为本章实例额外提供了 Cordova 对客户端进行打包的解决方案，可以快速生成一个 APK 文件（目前只支持 Android 端）。考虑到 Android 平台碎片化的问题，在本项目中引入 Crosswalk 来提供一个统一的 WebView，用以解决 WebRTC 的兼容问题，安装命令如下：

```
// 默认安装版本太低，要指定最新版本
cordova plugin add cordova-plugin-crosswalk-webview@2.3.0
```

除了兼容问题，还要对项目工程做一些调整，如下所示。

(1) 在 client 目录中找到 node_modules 目录，找到“msr/MediaStreamRecorder.js”文件，注释掉该文件中的如下代码：

```
// if (typeof location !== 'undefined') {
//   if (location.href.indexOf('file:') === 0) {
//     console.error('Please load this HTML file on HTTP or HTTPS.');
```

这是为了使项目在 file 协议下也能运行。

(2) client 目录中的代码编译完成后，把“client/build”目录中的文件全部复制到“client/www”目录中，并且将“www/index.html”中引用的静态资源的路径起始位置改为根目录。

读者可以在项目目录“platform/android/build/outputs/apk”下获取最终打包产生的 APK 文件，Cordova 的使用请参看“第 10 章 混合式开发实战”相关内容。

14.2 直播平台功能预览

在搭建直播平台之前要确定直播平台所包含的功能以及需要的技术。参考目前主流的直播 APP，通常包含以下几个部分：

- 注册、登录（支持第三方登录）
- 用户（基本信息、等级）
- 社交（动态、关注、聊天）
- 直播（房间、发起直播、观看直播、录制回顾）

- 推广（热门、附近、地区、频道等）
- 消费（金币、充值、礼物商城）

由此可见，直播平台是一个业务复杂度相对较高的项目。本章实例主要完成平台最核心的直播功能，即发起直播和观看直播。

14.2.1 直播流程

直播的流程由采集、推流、编码、拉流、解码、播放等环节组成，如图 14.3 所示。



图 14.3 直播流程

- 采集：发起端通过 Navigator 对象的 `getUserMedia` 方法调用本地摄像头采集视频流。
- 推流：发起端将视频流通过 WebSocket 上传到服务器。
- 编码：服务器使用 FFmpeg 视频框架将视频流编码为 TS 格式储存。
- 拉流：观看端通过 HLS 从服务器拉取编码后的音视频流。
- 观看端解码音视频流。
- 观看端播放解码后的音视频流。

14.2.2 直播核心页面

核心功能的实现只需要以下三个页面。

- 首页：可发起直播及显示正在进行的直播列表。

- 发起直播页面：设置视频参数、显示弹幕等。
 - 观看直播页面：播放直播、添加弹幕、显示弹幕。
- 页面间的交互如图 14.4 所示。

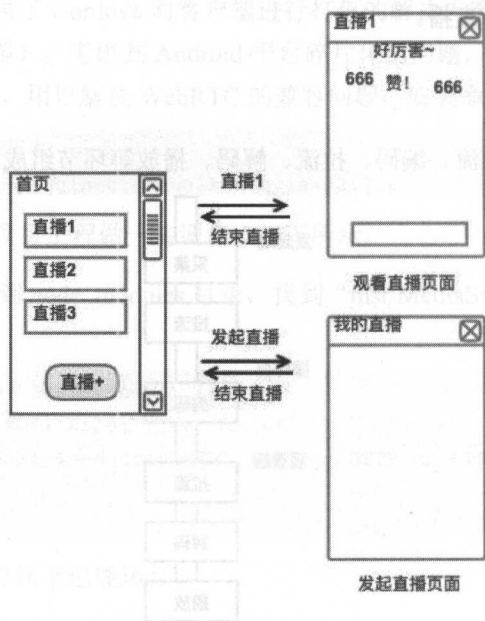


图 14.4 页面间交互关系

14.3 页面的布局 and 结构

本节将介绍页面的相关布局 and 结构。开发的功能及细节会随着本节的内容展开并最终确定下来。

14.3.1 首页

首页结构及相关功能如下。

(1) 直播列表：显示正在进行中的直播，列表中的每个项目包含直播的封面图片、标题和当前的观看人数；

(2) “创建直播”按钮：单击后跳转到发起直播页面，开始新的直播。

最终实现效果如图 14.5 所示。

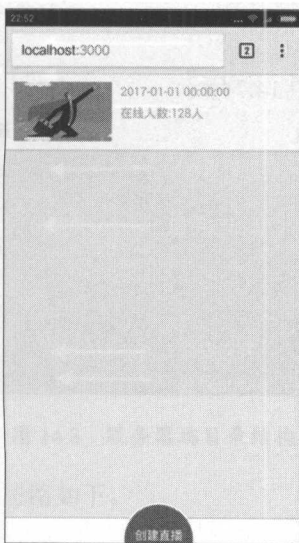


图 14.5 首页

14.3.2 发起直播页面

发起直播页面结构及相关功能如下：

- (1) 录制的视频实时预览并全屏显示在页面最底层。
- (2) 弹幕层显示在视频上层，实时显示参与者发出的评论，评论按时间顺序在屏幕某一区域内，从右至左移动。
- (3) 观看直播的实时人数显示在页面的左上角。
- (4) 按钮工具条显示在页面的右侧，包含控制直播、结束直播、切换前/后摄像头、静音等功能，并在用户单击 3 秒后自动隐藏，触碰屏幕唤起。
- (5) 页面底部的弹幕输入框可供输入弹幕，单击“发送弹幕”按钮发送弹幕内容至服务器，同时文字出现在弹幕层中。

页面最终实现效果如图 14.6 所示。

14.3.3 观看直播页面

观看直播页面的结构与发起直播页面类似，但按钮工具条有所不同，观看直播页面包含退出

直播、切换全屏、静音等功能。

观看直播页面最终实现效果如图 14.7 所示。

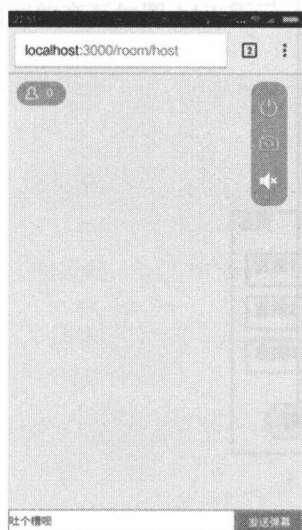


图 14.6 发起直播页面

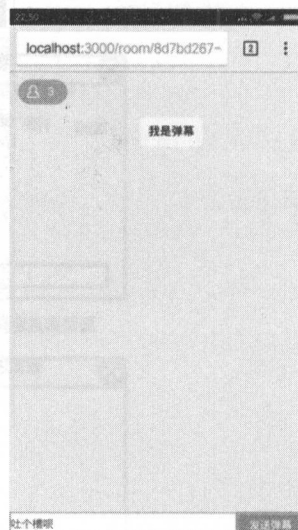


图 14.7 直播观看页面

14.4 搭建 WebRTC 服务端——Koa

本节将介绍如何搭建直播应用的服务器端以及 API 接口的设计。服务器端采用 Koa 框架实现本实例所需接口。

首先，创建目录 server，并且执行命令“npm init”初始化项目。然后安装项目所需依赖，命令如下：

```
npm install koa koa-bodyparser koa-router koa-static
```

文件目录结构如图 14.8 所示。

项目以 MVC 架构为基础，然而因为采用前后端分离的结构，因此服务器端只提供所需要的 API，也就没有“View”这一层。

提示：由于 WebRTC 需要 HTTPS 的支持，本例中采用自签证书的方式实现 HTTPS。自签证书使用 OpenSSL，本节不作赘述。

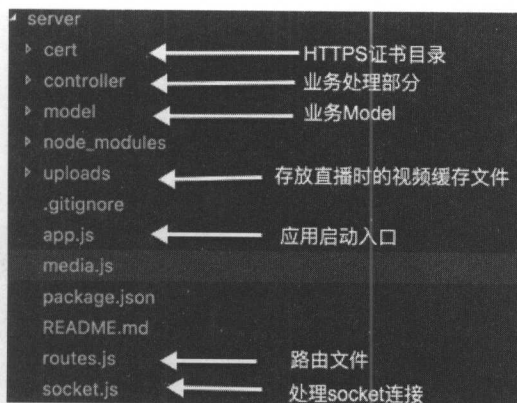


图 14.8 服务器端目录结构

视频直播采用 HLS 技术实现，思路如下。

- 采用 WebRTC 接口获取音视频信息。
- 采用 MediaRecorder 定时录制视频片段并上传到服务器。
- 服务器保存视频片段并生成 M3U8 格式文件。
- 其他参与直播的客户端直接采用 M3U8 播放直播视频。

由于视频需要采用 HLS 技术播放，因此需要输出 MPEG-TS 格式视频，然而录制的视频为 WebM 格式，所以需要在服务器端对视频源进行转码，转码采用 FFmpeg 组件。

提示：HLS 是由 Apple 提出的一种基于 HTTP 协议的在线视频直播解决方案，由播放列表文件（格式为 M3U8）和视频片段（格式为 TS）组成。

首先安装 FFmpeg。FFmpeg 提供了 Windows 和 Mac 的预编译包，直接安装就可以使用。如果服务器是 Linux 系统，比如 CentOS，则需要采用源码的方式进行编译安装。具体请参考官方说明文档 <https://trac.ffmpeg.org/wiki/CompilationGuide/Centos>。

在 Node.js 中，“fluent-ffmpeg”模块提供了对 FFmpeg 组件的封装，安装命令如下：

```
npm install fluent-ffmpeg
```

至此，服务器端环境已准备好。在本例中需要实现如下接口：

- 注册用户、获取用户信息
- 获取直播频道列表
- 添加直播频道
- 获取某一频道信息

- 开始直播
- 停止直播

提示: 同直播相关的接口, 将在下一节中介绍。

首先在 controller 目录下, 创建 Controller 的基类 base.js, 代码如下:

```
01 const Router = require("koa-router")
02 class Controller{
03     constructor(){ // 一个 Controller 对应一个子路由
04         this.routes = new Router()
05     }
06     renderJSON(context, data){ // 封装 renderJSON 方法, 输出 API 数据
07         context.set('Content-Type', 'application/json')
08         context.body = data
09     }
10     route(router, path){ // 将子路由挂载到主路由上
11         router.use(path, this.routes.routes())
12     }
13 }
14 module.exports = Controller
```

基类封装了基础的 renderJSON 方法, 并且定义了子路由和挂载到父路由的方法。这样, 可以将路由信息定义在 Controller 的内部。

接下来, 分别定义用户和频道相关的 Controller。首先定义用户相关的 API, 在 controller 目录下建立文件 user.js, 并定义用户相关 Controller 的相关接口, 代码如下:

```
01 const user = require("../model/user") // 引入用户 Model
02 const Base = require("./base") // 引入 Controller 基类
03 class User extends Base { // 定义用户 Controller, 继承自基类
04     constructor() {
05         super() // 调用基类的构造器
06         this.add()
07         this.getId()
08     }
09     add() { // 添加用户
10         this.routes.post('/', (context, next) => {
11             let item = context.request.form // 获取表单提交的数据
12             let u = user.add(item) // 调用 model 中的方法添加用户
13             this.renderJSON(context, { // 输出操作结果
14                 status: 0,
```

```

15         data: u
16     })
17 })
18 }
19 getById() { // 获取某一用户的信息
20     this.routes.get('/:id', (context, next) => { // 定义了子路由为"/:id"
21         let id = context.params.id // 获取 URL 地址中传入的用户 ID
22         let u = user.getUser(id) // 调用 model 的方法，获取某个用户信息
23         this.renderJSON(context, { // 输出结果
24             status: 0,
25             data: u
26         })
27     })
28 }
29 }
30 module.exports = (router) => { // 对外暴露方法，挂载 controller
31     new User().route(router, '/user') // 将子路由挂载到"/user"上
32 }

```

本例利用 koa-router 的功能在 Controller 层内部定义子路由，调用父路由的 use 方法，将子路由挂载到父路由上。通过这种方式提高了代码的可读性，降低了模块之间的耦合度。

上述代码中对数据的真实操作并没有放在 Controller 层，而是定义在相应的 Model 中。用户相关 Model 的代码如下：

```

01 const users = [] // 定义全局数组，用户信息将保存在该数组中
02 const uuid = require("node-uuid") // 引入 uuid 组件，生产唯一 ID
03 module.exports = {
04     add(user) { // 添加用户
05         user.id = user.id || uuid.v4() // 产生用户 ID
06         users.push(user) // 直接将用户保存在全局数组中
07         return user
08     },
09     remove(id) { // 删除用户
10         let index = users.findIndex((item) => {
11             return item.id === id
12         })
13         users.splice(index, 1)
14     },
15     getUser(id) { // 获取用户信息
16         return users[id]
17     }

```



```
18 }
```

本例仅将用户数据保存在内存中。在实际的项目中，可以将数据保存到 MongoDB 或者 MySQL 等数据库中。限于篇幅，本例不做具体介绍。

上述代码简要地介绍了用户相关操作的代码，接下来需要将用户相关 Controller 挂载到主路由中，这样这部分 API 就可以被外部所访问。在 Controller 目录下创建 index.js 文件，代码如下：

```
const user = require("../user") // 引入用户的 Controller
module.exports = (router) => {
  user(router) // 通过传递进来的主路由信息，将子路由挂载到主路由
}
```

这样，用户相关的 API 就定义完成了。接下来，介绍本项目中最核心的频道相关的 API，代码如下：

```
01 const Base = require("../base") // 引入基类
02 const channel = require("../model/channel.js") // 引入频道 model
03 class Channel extends Base {
04   constructor() {
05     super()
06     this.list()
07     this.add()
08     this.get()
09   }
10   list() { // 获取频道列表
11     this.routes.get("/", (context, next) => {
12       this.renderJSON(context, {
13         status: 0,
14         data: channel.get()
15       })
16     })
17   }
18   add() { // 添加频道
19     this.routes.post("/", (context, next) => {
20       let item = JSON.parse(context.request.body)
21       channel.add(item)
22       this.renderJSON(context, {
23         status: 0
24       })
25     })
26   }
```

```

27  get() { // 获取某一频道
28      this.routes.get("/:id", (context, next) => {
29          let c = channel.getOne(context.params.id)
30          if (c) {
31              this.renderJSON(context, {
32                  status: 0,
33                  data: c
34              })
35          } else {
36              this.renderJSON(context, {
37                  status: -1,
38                  message: `the channel with ${context.params.id} not found`
39              })
40          }
41      })
42  }
43  }
44  module.exports = (router) => { // 将子路由挂载到主路由
45      new Channel().route(router, '/channel')
46  }

```

频道 Model 和用户 Model 相似，都是将数据保存在全局数组中，读者可以查看源代码中的具体实现。

在项目根目录下，创建一个 routes.js 文件，此文件引入 Controller 中的路由以及其他的子路由。定义主路由，加载子路由，并返回路由信息给 Koa 调用，代码如下：

```

const Router = require("koa-router") // 引入 Koa 路由组件
const controller = require("../controller") // 引入 controller 的 index 文件
const route = new Router() // 创建主路由
controller(route) // 挂载 controller 中的子路由
module.exports = route // 返回路由信息给 Koa 调用

```

在 app.js 文件中，引入定义的路由信息和相关中间件，创建 Koa 应用，代码如下：

```

const Koa = require("koa") // 引用 Koa 对象
const app = new Koa() // 创建 Koa 实例应用对象
const fs = require("fs") // 引入 fs 对象，操作文件
const path = require("path") // 引入 path 对象，处理路径
const http = require("http") // 引入 HTTP 对象，创建 HTTP 服务
const https = require("https") // 引入 HTTPS 对象，创建 HTTPS 服务
const bodyParser = require("koa-bodyparser") // 引入 bodyparser 插件，解析 HTTP Body
const route = require("../routes") // 引入前面介绍的主路由信息

```

```

app.use(bodyParser()) // 使用 bodyparser 中间件
app.use(route.routes()) // 使用 router 中间件
app.use(route.allowedMethods())
let callback = app.callback() // 得到 HTTP 请求的 handler
const server = http.createServer(callback) // 依据 handler 创建 HTTP 服务
const { PORT = 4412, HTTPS_PORT = 4413 } = process.env
let httpsServer = https.createServer({ // 依据 handler 和证书创建 HTTPS 服务
  key: fs.readFileSync(path.resolve(__dirname, './cert/private.key')),
  cert: fs.readFileSync(path.resolve(__dirname, './cert/pub.crt'))
}, callback)
server.listen(PORT) // 监听 HTTP 端口
console.log(`http start at ${PORT}`)
httpsServer.listen(HTTPS_PORT) // 监听 HTTPS 端口
console.log(`https start at ${HTTPS_PORT}`)

```

至此,实例的 API 初步创建完毕,将服务部署到服务器后即可使用。限于篇幅和复杂度,本例并没有使用数据库服务。当然,读者可以修改 Model 中的实现,将保存在全局数组中的数据改为存放在数据库中。在实际的项目中部署 Node.js 时,为了提升系统的扩展性,可以使用 PM2 启动并开启 Cluster 模式。

14.5 实现多人在线直播功能

本节将介绍关于使用 WebSocket 实现直播功能的细节。由于可以把参与的用户当成一个群组,在 Socket 实现时,直接采用“namespace”的方式,把每个直播频道独立为一个“namespace”。

基于这样的思路,在开始直播的时候需要创建 Socket 的“namespace”。由于每个“namespace”的功能几乎相同,所以可以直接创建一个类(Class)来处理直播中 Socket 的逻辑。不同的“namespace”使用不同的实例来处理。

在 Socket 类中主要有如下三个逻辑。

- 接收客户端的身份信息,标注当前用户信息。
- 接收直播者发出的视频数据,转码后保存到本地磁盘。
- 产生 M3U8 播放列表。

首先,介绍这个类的基础定义以及标注用户信息的逻辑。核心代码如下:

```

01 class Socket {
02   constructor(id) { // 根据传入的频道 ID 来创建对应的 socket 实例

```



```

03     this.id = id
04     this.ns = instance.of(`/${id}`) // 创建或获取名为频道 ID 的 socket 命名空间
05     if(this.ns.init){                // 如果命名空间已经创建
06         C.ready(this.id)             // 此时是重新开始直播，更新频道状态
07         return
08     }
09     this.ns.init = true               // 标记当前的命名空间已经创建过了
10     this.init()                      // 初始化类
11 }
12 init() {
13     this.ns.on("connection", (socket) => { // 在命名空间连接时
14         socket.on("login", (data) => {     // 当接受到 login 消息时，注册用户信息
15             socket.user = data
16         })
17     })
18 }
19 }

```

在实例化 **Socket** 类时，上述代码会创建一个名为传入的频道 ID 的命名空间，并在 **Socket** 发生连接时，监听登录消息，将客户端传递的用户信息记录到 **Socket** 连接对象上，以便后续逻辑调用。

采用“**socket.io-stream**”组件以 **Stream** 的方式对 **Socket** 发送的数据进行处理，将客户端发送来的视频数据进行转码之后保存到磁盘，代码如下：

```

ioStream(socket).on("upload", (stream) => {
    let filename = `${this.id}-${this.sequence}.ts` // 生成视频片段的文件名
    let filePath = path.resolve(__dirname, `./uploads/${filename}`)
    ffmpeg(stream).videoCodec('libx264')
    .audioCodec('aac')
    .addOption('-mpegts_copyts', 051)
    .addOption('-strict', -2)
    .format('mpegts')
    .addOutputOption('-output_ts_offset', this.duration)
    .on('error', (err) => {
        debug('an error happened: ' + err.message); // 输出错误信息
    }).save(filePath) // 保存视频片段到磁盘
    .on('end', () => {
        /* 省略，参考源代码 */
    })
})

```


上述代码中的核心代码采用 FFMpeg 组件对视频进行了转码。这里使用的基本方法如下。

- **videoCodec**: 指定视频编码器, 依据 HLS 的要求, 本处采用 “libx264” 视频编码器。
- **audioCodec**: 指定音频编码器, 本处采用 “ACC” 音频编码器。
- **format**: 指定输出的格式, 本处采用 “MPEG-TS” 格式。
- **addOutputOption**: 指定输出参数, 本处设置 “-output_ts_offset” 来调整当前视频片段在总视频中时长的偏移量。偏移量从 0 开始, 下一个片段的偏移量为前面视频片段的总时长。
- **adoption**: 设置参数, 为了提升后面时长计算的准确度。

参数中的 “-output_ts_offset” 要求提供每个视频片段的时长信息。可以采用 `ffmpeg` 对象的 `fprobe` 方法获取该信息, 代码如下:

```
ffmpeg.ffprobe(filePath, (err, data) => {
  if (err) debug(err) // 输出出错信息
  this.duration += data.format.duration // 计算当前视频的总时长
  this.addVideo(`/uploads/${filename}`, data.format.duration) // 添加视频
  this.sequence++ // 增加视频序号
  // 已经缓存了几个片段之后, 才开始向客户端通知可以播放
  if (this.sequence === MAX_VIDEO_FILES) {
    C.ready(this.id)
    socket.emit("videoReady")
  }
})
```

通过以上代码获取当前保存的视频文件的视频信息, 并得到时长。依次计算总时长, 便于在某个视频进行转码时设置下一视频的 “-output_ts_offset” 参数。直播开始时, 视频数据还没有处理完毕, 因此此时客户端无法播放。为解决这一问题需要统一处理视频源端 `ready` 状态。当已经保存了最大视频片断数之后, 修改 `Channel` 实体的 `ready` 状态, 并给用户发送 “videoReady” 消息。当浏览器获知当前视频已经 `ready`, 才会直接播放视频, 否则等到 “videoReady” 事件触发之后才开始播放视频。

上述代码中的 `addVideo` 方法定义在 `Channel` 模块中, 代码如下:

```
addVideo(video, duration) {
  C.addVideo(this.id, { // 调用 channel model 的方法添加视频片段
    sequence: this.sequence,
    video,
    duration
  })
}
```

`addVideo` 方法将新的片段添加到片段列表中, 以便继续生成播放列表文件, 该方法代码如下:

```
addVideo(id, video) {
  let item = C.getOne(id) // 根据 id 获取频道信息
  if (item && item.videos) { // 检查是否获取到了频道信息
    let videos = item.videos
    if (videos.length >= MAX_VIDEO_FILES) { // 如果当前缓存的片断数超出最大数
      let file = path.resolve(__dirname, `..${videos[0].video}`)
      fs.unlink(file, (err, data) => { // 删除第一个文件
        if (err) {
          debug(err)
        } else {
          debug(`remove file ${file}`)
        }
      })
      videos.splice(0, 1) // 删除第一个片段数据
    }
    videos.push(video) // 在末尾添加新片段
  }
}
```

直播列表中需要实时展示在线人数, 该数据可以通过监听 `connection` 和 `disconnect` 事件来计算, 代码如下:

```
const online = () => {
  this.ns.clients((err, cs) => { // 获取当前连接到 namespace 上的客户端数
    let onlines = cs.length // 在线人数即连接的客户端数
    this.ns.emit("online", onlines) // 发送 socket 消息通知客户端
    C.getOne(this.id).onlines = onlines // 将在线数据同步到 Channel 模块中
  })
}
```

当直播者的 Socket 连接断开时, 直播结束。为避免直播者关闭浏览器时服务器端的状态仍处于更新状态, 代码如下:

```
01 socket.on('disconnect', () => {
02   let channel = C.getOne(this.id)
03   if (!socket.user.id === channel.owner) { // 如果当前用户不是直播者, 不处理
04     return
05   }
06   C.done(this.id) // 标记频道状态为直播完成
07   this.videos.forEach((item) => { // 删除所有缓存的文件
08     let file = path.resolve(__dirname, `.${item.video}`)
```

```

09     fs.unlink(file, (err) => {
10         if (err) {
11             debug(`delete file: ${file} error`)
12         }
13     })
14 })
15 })

```

在 Socket 模块中定义根据频道 ID 创建 Socket 实例的方法，以便频道模块调用，代码如下：

```

const create = (id) => {
    new Socket(id)           // 直接以频道 ID 为参数，实例化 Socket 对象
}

```

至此，Socket 模块中的主要代码介绍完毕。该模块处理 Socket 的数据通信，并将直播相关的状态更新到频道模块中。接下来介绍频道模块中与直播相关的方法。

Channel 控制器中定义了 begin 和 end 方法，表示开始直播和结束直播，代码如下：

```

01 begin() {                                // 开始直播调用此 API
02     this.routes.get('/:id/begin', (context, next) => {
03         let id = context.params.id
04         socket.create(id)                 // 调用 Socket 中封装的 create 方法创建 Socket 实例
05         this.renderJSON(context, {
06             status: 0
07         })
08     })
09 }
10 end() {                                    // 停止直播调用此 API
11     this.routes.get('/:id/end', (context, next) => {
12         let id = context.params.id
13         channel.done(id)                  // 结束直播
14         this.renderJSON(context, {
15             status: 0
16         })
17     })
18 }

```

客户端利用直播源观看直播。调用频道模块中的 getM3U 方法获取直播源，代码如下：

```

playList() {
    this.routes.get('/:id/playlist', (context, next) => {
        let id = context.params.id
        context.set('Content-Type', 'application/x-mpegURL')
    })
}

```



```

context.body = channel.getM3U(id) // 输出频道模块中生成的 M3U8 信息
})
}

```

在频道模块中，调用从 addVideo 方法得到的视频片断来生成播放列表，代码如下：

```

01 getM3U(id) {
02     let item = C.getOne(id) // 按照频道 ID 获取频道实体
03     let videos = item.videos // 获取频道下的视频片段
04     let writer = m3u8.httpLiveStreamingWriter(); // 调用 m3u 组件，创建 m3u8 生成器
05     writer.version(3);
06     writer.targetDuration(4);
07     writer.allowCache(false);
08     if (videos.length) { // 检测是否有视频片段
09         writer.mediaSequence(videos[0].sequence) // 添加 Sequence 信息
10     }
11     writer.write();
12     videos.forEach((item) => { // 将每个视频片段加入到列表中
13         writer.file(`${item.video}`, item.duration, `${this.id}-${item.sequence}`)
14     })
15     let str = writer.toString()
16     debug(`playlist: ${str}`);
17     return str
18 }

```

至此，频道 Controller 中的开始直播、停止直播和获取播放列表介绍完毕。浏览器端可以通过这些 API 处理直播相关业务。

14.6 实现弹幕客户端与服务端通信

弹幕是视频直播网站经常使用的交互手段之一，实现弹幕的方式有很多种，根据实现方式的不同，弹幕有实时与非实时之分，本节介绍的是使用 WebSocket 实现实时弹幕的方法。

14.6.1 客户端与服务端通信的过程

要实现弹幕功能，首先要创建和直播频道对应的 WebSocket 连接，客户端经由 WebSocket 连接将弹幕信息发送到服务器，再由服务器转发到所有与直播频道关联的 WebSocket 连接中，客户端在监听到服务端推送的信息后，将信息显示在视频上方并滚动播放。

14.6.2 客户端代码设计——React

在客户端,为了方便在直播间与转播间分别使用弹幕,需要将弹幕功能与视频直播功能分离,并将弹幕功能抽象为 Barrage 组件。Barrage 组件包含 BarrageInput 和 BarrageList 两个子组件,分别对应弹幕的输入和弹幕的展示两个功能。

当用户输入弹幕信息并单击“发送”按钮或回车键后,BarrageInput 子组件会调用父组件 Barrage 提供的 onBarrageSend 方法传送弹幕信息,实现代码如下:

```
01 import React, { Component } from 'react'
02 import ReactDOM from 'react-dom';
03 export default class BarrageInput extends Component {
04     constructor(props) {
05         super(props)
06     }
07     componentDidMount() {
08         let barrage = ReactDOM.findDOMNode(this.refs.barrageMessage) // 获取输入框
09         barrage.addEventListener('keydown', (e)=>{ // 监听按键
10             if (e.keyCode === 13) this.sendBarrage() // 回车发送
11             }, false)
12     }
13     sendBarrage() {
14         let barrage = ReactDOM.findDOMNode(this.refs.barrageMessage) // 获取输入框
15         if (barrage.value) {
16             this.props.onBarrageSend(barrage.value) // 发送弹幕
17             barrage.value = '' // 重置输入框
18         }
19     }
20     render() {
21         return (
22             <div className="barrage-input-container">
23                 <input type="text" className="barrage-input-text" ref="barrageMessage"
placeholder="吐个槽呗" maxLength="10"/>
24                 <input type="button" value="发送弹幕" onClick={e => this.sendBarrage(e)}
className="barrage-input-submit"/>
25             </div>
26         )
27     }
28 }
```

BarrageList 子组件接收父组件 Barrage 提供的 Props。当父组件 Barrage 的 Props 发生改变时,

更新渲染并通过 `ReactCSSTransitionGroup` 附加对应的 CSS 样式名，以加载事先已经编写好的 CSS 弹幕动画，实现代码如下：

```
01 import React, { Component } from 'react'
02 import ReactCSSTransitionGroup from 'react-addons-css-transition-group'
03 export default class BarrageList extends Component {
04   constructor(props) {
05     super(props)
06     this.state = {barrages: this.props.barrages} // 初始化 state
07   }
08   componentWillReceiveProps(nextProps) {
09     this.setState({barrages: nextProps.barrages}); // 接收到 props 后改变 state
10   }
11   render() {
12     const barrages = this.state.barrages ? this.state.barrages.map((barrage, index) => (
13       <div key={index} className={`barrage-${index % 8}`}>
14         {barrage}
15       </div>
16     )) : []
17     return (
18       <div className="barrages">
19         <ReactCSSTransitionGroup // React 动画组件
20           transitionName="barrage"
21           transitionEnterTimeout={4000}
22           transitionLeave={false}>
23           {barrages}
24         </ReactCSSTransitionGroup>
25       </div>
26     )
27   }
28 }
```

`Barrage` 组件首先按照直播频道对应的标识符创建 `WebSocket` 连接，并在这个 `WebSocket` 连接中监听对应的消息事件。当接收到从服务端传送的消息后，调用 `addBarrage` 方法更新 `State` 并传送到子组件 `BarrageList` 中。当 `Barrage` 父组件接收到子组件 `BarrageInput` 传送过来的弹幕信息后，则会调用 `sendBarrage` 方法将弹幕信息通过 `WebSocket` 发送到服务器端，代码如下：

```
01 import config from '../config'
02 import io from 'socket.io-client'
03 import React, { Component } from 'react'
04 import BarrageList from './BarrageList'
```

```

05 import BarrageInput from './BarrageInput'
06 export default class Barrage extends Component {
07   constructor(props) {
08     super(props)
09     if(this.props.channel){
10       let cid = this.props.channel.id
11       this.socket = io(`${config.httpServer}/${cid}`) // 创建 WebSocket 连接
12       this.socket.on('new message', (data)=>{ // 监听 WebSocket 消息
13         this.addBarrage(data);
14       });
15     }
16     this.state = {barrages: []} // 初始化 barrages
17   }
18   addBarrage(item) {
19     const newBarrage = this.state.barrages.concat([item.message]);
20     this.setState({barrages: newBarrage}); // 设置 barrages
21   }
22   sendBarrage(barrage) {
23     this.socket && this.socket.emit('new message', barrage); // 发送弹幕到服务器
24   }
25   render() {
26     return (
27       <div>
28         <BarrageList barrages={this.state.barrages} />
29         <BarrageInput onBarrageSend={this.sendBarrage.bind(this)} />
30       </div>
31     )
32   }
33 }

```

最后只要在直播间和转播间的组件中加入 **Barrage** 组件并传入直播频道标识即可,弹幕的客户端代码就完成了。

14.6.3 服务端代码设计

服务端代码相对简单,只需将从客户端接收到的消息在所有对应的 **WebSocket** 连接中进行转发即可,代码如下:

```

socket.on('new message', (data) => { // 监听客户端发送来的消息
  this.ns.emit('new message', { // 将消息转发

```



```
message: data
  })
})
```

至此，直播平台的弹幕功能就完成了。弹幕客户端代码中所使用的 CSS 样式不在此处赘述，具体请参考源代码或 GitHub 中的项目 iKtalk，地址为 <https://github.com/ikcamp/iKtalk>。

14.7 本章小结

本章介绍了如何使用 Web 技术开发一个直播应用，涉及的新技术有 WebRTC、WebSocket、React、FFmpeg 等。读者可以对其中感兴趣的技术点进行深入研究。目前，Web 端发起直播还不是主流选择。原因有两个：

- 一是性能问题；
- 二是 WebRTC 只能被部分浏览器支持。

WebRTC 提供了在 Web 端发起视频会议的能力，本例选择了 WebRTC 的部分协议能力来采集直播发起方的视频，将转码和服务压力放在 Node.js 服务器端，借以提供更稳定的服务。在实际项目中，开发者需要根据视频服务的种类和规模选择合适的技术，在面对大并发的情况下，通过例如 CDN 缓存视频信息等手段进行性能优化。

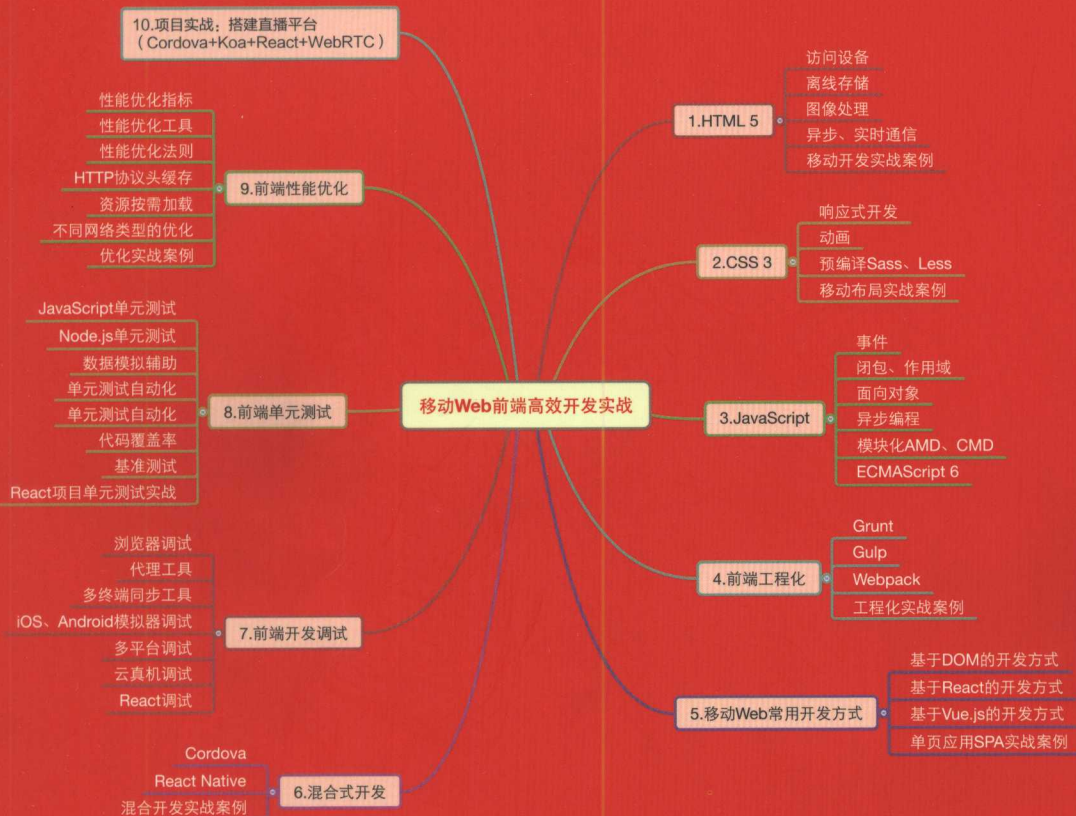
好书分享



反馈意见或投稿，联系编辑：

邮箱：dongying@phei.com.cn

微信：yingzidd



上架建议：前端开发

ISBN 978-7-121-32481-9



9 787121 324819 >

定价：89.00元



博文视点Broadview



@博文视点Broadview



责任编辑：董 英
封面设计：李 玲